29th European Conference on Object-Oriented Programming

ECOOP'15, July 5–10, 2015, Prague, Czech Republic

Edited by

John Tang Boyland

Editor

John Tang Boyland
Department of EE & Computer Science
University of Wisconsin-Milwaukee, USA
`boyland@uwm.edu`

ACM Classification 1998
D.1.5 Object-oriented Programming

ISBN 978-3-939897-86-6

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/978-3-939897-86-6.

Publication date
July 2015

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.

License
This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0): `http://creativecommons.org/licenses/by/3.0/legalcode`.

In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.ECOOP.2015.i

ISBN 978-3-939897-86-6 **ISSN 1868-8969** **http://www.dagstuhl.de/lipics**

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.

Editorial Board

- Susanne Albers (TU München)
- Chris Hankin (Imperial College London)
- Deepak Kapur (University of New Mexico)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Catuscia Palamidessi (INRIA)
- Wolfgang Thomas (*Chair*, RWTH Aachen)
- Pascal Weil (CNRS and University Bordeaux)
- Reinhard Wilhelm (Saarland University)

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

Preface
John Tang Boyland ix

Artifacts
Camil Demetrescu and Matthew Flatt xi

Organization xiii

External Reviewers xv

List of Authors xvii

Abstracts of Invited Talks

Object-Oriented Programming without Inheritance
Bjarne Stroustrup 1

Programming in the Large for the Internet of Things
Jong-Deok Choi 2

Software Verification “Across the Stack”
Alexander J. Summers 3

Gradual Typing

Towards Practical Gradual Typing
Asumu Takikawa, Daniel Feltey, Earl Dean, Matthew Flatt, Robert Bruce Findler, Sam Tobin-Hochstadt, and Matthias Felleisen 4

TreatJS: Higher-Order Contracts for JavaScript
Matthias Keil and Peter Thiemann 28

Trust, but Verify: Two-Phase Typing for Dynamic Languages
Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala 52

Implementation

Concrete Types for TypeScript
Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek 76

Simple and Effective Type Check Removal through Lazy Basic Block Versioning
Maxime Chevalier-Boisvert and Marc Feeley 101

Loop Tiling in the Presence of Exceptions
Abhilash Bhandari and V. Krishna Nandivada 124

LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Objects

Transparent Object Proxies for JavaScript
Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann 149

A Theory of Tagged Objects
Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin 174

Brand Objects for Nominal Typing
Timothy Jones, Michael Homer, and James Noble 198

Analysis I

Access-rights Analysis in the Presence of Subjects
Paolina Centonze, Marco Pistoia, and Omer Tripp 222

Variability Abstractions: Trading Precision for Speed in Family-Based Analyses
Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski 247

Developer Assistance

Optimization Coaching for JavaScript
Vincent St-Amour and Shu-yu Guo 271

PerfBlower: Quickly Detecting Memory-Related Performance Problems via Amplification
Lu Fang, Liang Dou[1]*, and Guoqing Xu* 296

Hybrid DOM-Sensitive Change Impact Analysis for JavaScript
Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman 321

Types

Intensional Effect Polymorphism
Yuheng Long, Yu David Liu, and Hridesh Rajan 346

Type Inference for Place-Oblivious Objects
Riyaz Haque and Jens Palsberg 371

Asynchronous Liquid Separation Types
Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis 396

Parallelism

The Eureka Programming Model for Speculative Task Parallelism
Shams Imam and Vivek Sarkar 421

[1] Work was done while the author visited UC Irvine during 2013–2014.

Cooking the Books: Formalizing JMM Implementation Recipes
Gustavo Petri, Jan Vitek, and Suresh Jagannathan 445

Defining Correctness Conditions for Concurrent Objects in Multicore Architectures
Brijesh Dongol, John Derrick, Lindsay Groves, and Graeme Smith 470

Empirical Studies

The Love/Hate Relationship with the C Preprocessor: An Interview Study
Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi 495

The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript
Michael Pradel and Koushik Sen 519

Abstraction

A Pattern Calculus for Rule Languages: Expressiveness, Compilation, and Mechanization
Avraham Shinnar, Jérôme Siméon, and Martin Hirzel 542

Global Sequence Protocol: A Robust Abstraction for Replicated Shared State
Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich . 568

Streams à la carte: Extensible Pipelines with Object Algebras
Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis .. 591

Verification

Lightweight Support for Magic Wands in an Automatic Verifier
Malte Schwerhoff and Alexander J. Summers 614

Modular Verification of Finite Blocking in Non-terminating Programs
Pontus Boström and Peter Müller 639

Modular Termination Verification
Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper 664

Analysis II

Framework for Static Analysis of PHP Applications
David Hauzar and Jan Kofroň 689

Adaptive Context-sensitive Analysis for JavaScript
Shiyi Wei and Barbara G. Ryder 712

Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity
Changhee Park and Sukyoung Ryu 735

Preface

This volume contains 31 accepted papers (of 136 submitted) of the 29th ECOOP, ECOOP'15 which will take place in Prague in July of 2015. Each submitted paper was assigned at least three program committee (PC) reviewers. If a PC member was an author, then the paper was assigned a minimum of five PC reviewers. Over twenty additional external reviewers wrote reviews of papers where we needed more expertise. External reviewers had to be approved by the program chair. Altogether 450 reviews were produced.

Papers authored by PC members were given extra scrutiny. Along with the extra reviews, the papers were decided upon before the PC meeting. Each paper had to be clearly acceptable with at least one champion and no objections to inclusion. Of the fifteen PC papers, only three were accepted. Several of the rejected papers would probably have been accepted in normal circumstances.

The program committee met in at ETH in Zürich to decide on the non-PC papers that had at least one positive review. We discussed these 80 papers in eight groups of ten papers each. After the discussion, each paper had a PC member tasked with writing up a summary of the discussion for the authors.

This proceedings is the first one to be published by the Leibniz International Proceedings in Informatics (LIPIcs). We appreciate the long partnership with Springer Verlag, who not only published the proceedings for most of the previous instances of ECOOP, but also provided in-kind funds to the authors of the best paper. Moving to LIPIcs allows ECOOP to provide "Gold Open Access" to accepted papers, which not only raises visibility, but also meets the requirements of increasingly many funding providers for open access. We look forward to a positive partnership with the Schloss Dagstuhl Leibniz-Zentrum für Informatik.

I would like to take this opportunity to thank Malte Schwerhoff and Marlies Weissert of ETH who were invaluable in handling the logistics of the meeting and to Peter Müller whose office provided funding and hosting of the event. I thank my own institution (University of Wisconsin-Milwaukee) for a reduced teaching schedule in the Spring semester of 2015, as well as providing a lunch for PC members. I also want to thank all the PC members for their hard work and thoughtful reviews. Last I would like to thank the ECOOP 2015 "Comfy Chair" Jan Vitek for his constant availability for questions and assistance.

John Tang Boyland

May, 2015

Artifacts

The ECOOP Artifact Evaluation (AE) process considers artifacts – software, data, proofs, videos, etc. – that are associated with published papers and that are independently submitted, reviewed, and accepted or rejected by an Artifact Evaluation Committee (AEC). The long-term goal of this process is to foster a culture of reproducibility of experimental results by considering software artifacts as first-class citizens, as well as enhancing the information provided to the community about research results. Artifacts are reviewed and accepted even if they cannot be made available to the public (e.g., because of confidentiality requirements or intellectual property difficulties), but the intent is that artifacts should be made available if possible. This year's ECOOP is the third edition to include AE, and similar processes continue to be adopted at other top conferences.

The AE process is similar to a paper-review process; artifacts are submitted by paper authors and evaluated by a committee based on individual assessments followed by a discussion among the reviewers. As is traditional, the ECOOP 2015 AEC members are all outstanding junior researchers. Each of the 17 AEC members reviewed 2-3 artifacts, and each artifact was evaluated by 3 members.

In the first phase, reviewers were asked to "kick the tires" of each artifact to check that it could be reviewed effectively. An author-response period followed immediately afterward. This phase ruled out corrupt artifact archive files and similar low-level problems that could easily be resolved with help from the authors.

In the second phase, the reviewers read the accepted papers, evaluated the associated artifacts with respect to the content and claims of the paper, and wrote evaluation summaries. In their artifact evaluations, reviewers focused on four key questions: (1) *Is the artifact consistent with the paper?* (2) *Is the artifact complete?* (3) *Is the artifact well documented?* and (4) *Is the artifact easy to reuse?* Each reviewer assigned an overall rating of "does not live up to expectations [raised by the paper]," "lives up to the expectations," or "exceeds expectations" for each artifact. In a virtual AEC meeting, the committee discussed those ratings and reviews and decided on acceptance or rejection for each artifact. During the discussion, all AEC members could see all reviews and discussions (except as proscribed by a conflict of interests), which allowed a calibration of the reviews across artifacts and reviewers.

Among the 31 papers accepted for ECOOP 2015, we received 15 artifacts for evaluation.[2] Of the submitted artifacts, the committee accepted 12 and rejected 3. A high acceptance rate seems natural for the AE process, since it covers only artifacts for papers that have already been accepted for publication. Currently, the AE process is not intended to influence paper submission, and independence is ensured by opening artifact submission only after paper notifications. As the AE process evolves, it is possible that the intent and application of AE influence will change.

Papers with accepted artifacts in this proceedings are marked with a rosette representing the seal of approval by the AEC. We are glad to note that all accepted artifacts are collected on the Dagstuhl Research Online Publication Server (DROPS) alongside the papers, and for the first time each artifact has its own DOI separate from its paper's DOI.

This year's AE process benefited greatly from the experience and advice of previous AEC organizers. We relied on the guidelines by Shriram Krishnamurthi, Matthias Hauswirth, Steve Blackburn, and Jan Vitek published in the foundational on-line article *Artifact Evaluation for*

[2] One accepted paper's artifact was ineligible for review, due to having an AEC co-chair as an author.

Software Conferences available at `http://www.artifact-eval.org`. The *Artifact Evaluation Artifact* effort by Steve Blackburn and Matthias Hauswirth, available at the address `http://evaluate.inf.usi.ch/artifacts/aea`, was also of inspiration. We thank the Program Chair John Boyland for his help and cooperation, and we particularly thank Jan Vitek for his continued involvement and advice. Thanks also to Eddie Kohler for his help with the HotCRP conference management software. Most significantly, we enthusiastically commend the AEC members for their diligent efforts. Finally, we thank all authors for packaging and documenting their artifacts for ECOOP 2015 and for making them publicly available; we believe that this extra step of publication is an invaluable service for the community.

Camil Demetrescu
Matthew Flatt

May, 2015

Organization

Comfy Chair

Jan Vitek (Northeastern University)

General Chairs

Tomas Kalibera
Pavel Kordik (Czech Technical University)

Program Chair

John Tang Boyland (University of Wisconsin-Milwaukee)

Program Committee

Stephanie Balzer (Carnegie Mellon University)
Walter Binder (University of Lugano)
Eric Bodden (Fraunhofer SIT and TU Darmstadt)
Viviana Bono (University of Torino)
Einar Broch Johnsen (University of Oslo)
Dave Clarke (Uppsala University and KU Leuvern)
Werner Dietl (University of Waterloo)
Danny Dig (Oregon State University)
Irene Finocchi (Sapienza University, Rome)
Christian Hammer (Saarland University)
Martin Hirzel (IBM Research)
Marieke Huisman (University of Twente)
Xuandong Li (Nanjing University)
Francesco Logozzo (Facebook)
Yi Lu (Oracle Labs)
Todd Millstein (University of California, Los Angeles)
Peter Müller (ETH Zurich)
Bruno Oliveira (University of Hong Kong)
Tamiya Onodera (IBM Research – Tokyo)
Pavel Parízek (Charles University in Prague)
Matthew Parkinson (Microsoft Research, UK)
Christoph Reichenbach (Goethe University)
Marco Servetto (Victoria University of Wellington)
Friedrich Steimann (Fernuniversität in Hagen)
T. Stephen Strickland (Brown University)
Mohsen Vakilian (Google)
Tom Van Cutsem (Alcatel-Lucent Bell Labs)
Harry Xu (University of California, Irvine)
Nobuko Yoshida (Imperial College London)

Artifact Evaluation Co-chairs

Camil Demetrescu (Sapienza University of Rome)
Matthew Flatt (University of Utah)

Artifact Evaluation Committee

Karim Ali (Technical University at Darmstadt)
Stefan Blom (University of Twente)
Lubomir Bulej (Charles University in Prague)
Nicolas Cardózo (Trinity College Dublin)
Fernando Chirigati (NYU Polytechnic School of Engineering)
Emilio Coppa (Sapienza University of Rome)
Paolo G. Giarrusso (University of Tübingen)
Raghavendra Kagalavadi (Oracle Labs)
Du Li (Carnegie Mellon University)
Sihan Li (University of Illinois at Urbana-Champaign)
Stefan Marr (Inria, France)
Philip Mayer (Ludwig Maximilians University Munich)
Cyrus Omar (Carnegie Mellon University)
Daniel Perelman (University of Washington)
Cosmin Radoi (University of Illinois)
Christophe Scholliers (Vrije Universiteit Brussel)
Wei Yang (University of Illinois at Urbana-Champaign)

External Reviewers

Jonathan Aldrich
Gavin Bierman
Robert Bocchino
Stefan Brunthaler
Satish Chandra
Tiago Cogumbreiro
Lu Fang
Yang Feng
Stephen Fink
Steve Freund
François Gauthier
Andy Gordon
Robert Grimm
Johannes Lerch
Padmanabhan Krishnan
Du Li
Stefan Marr
Ethan Munson
Khanh Nguyen
Kim Nguyen
Tien Nguyen
Dominic Orchard
Tomas Petricek
Andreas Rossberg
Andrew Santosa
Bernhard Scholz
Peter Sewell
Gregor Snelting
Matthew Staats
Tijs van der Storm
Emilio Tuoso
Jules Villard
Gulfem Savrun Yeniceri
Wei Zhang

29th European Conference on Object-Oriented Programming.
Editor: John Tang Boyland

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Authors

Jonathan Aldrich
Saba Alimadadi
Abhilash Bhandari
Aggelos Biboudis
Dragan Bosnacki
Pontus Boström
Claus Brabrand
Sebastian Burckhardt
Paolina Centonze
Maxime Chevalier-Boisvert
Benjamin Cosman
Earl Dean
John Derrick
Aleksandar S. Dimovski
Brijesh Dongol
Liang Dou
Lu Fang
Marc Feeley
Matthias Felleisen
Daniel Feltey
Robert Bruce Findler
Matthew Flatt
George Fourtounis
Manuel Fähndrich
Manuel Geffken
Rohit Gheyi
Lindsay Groves
Shu-yu Guo
Sankha Narayan Guria
Riyaz Haque
David Hauzar
Martin Hirzel
Michael Homer
Shams Imam
Suresh Jagannathan
Bart Jaobs
Ranjit Jhala
Timothy Jones
Matthias Keil
Johannes Kloos
Jan Kofro
Ruurd Kuiper
Christian Kästner
Joseph Lee
Daan Leijen
Yu David Liu
Yuheng Long
Rupak Majumdar
Flávio Medeiros
Ali Mesbah
Peter Müller
Sarah Nadi
V Krishna Nandivada
James Noble
Nick Palladinos
Jens Palsberg
Changhee Park
Karthik Pattabiraman
Gustavo Petri
Marco Pistoia
Alex Potanin
Michael Pradel
Jonathan Protzenko
Hridesh Rajan
Márcio Ribeiro
Gregor Richards
Barbara G. Ryder
Sukyoung Ryu
Vivek Sarkar
Andreas Schlegel
Malte Schwerhoff
Koushik Sen
Troy Shaw
Avraham Shinnar
Jerome Simeon
Yannis Smaragdakis
Graeme Smith
Vincent St-Amour
Alexander J. Summers
Asumu Takikawa
Peter Thiemann
Sam Tobin-Hochstadt
Omer Tripp
Viktor Vafeiadis
Panagiotis Vekris
Jan Vitek
Andrzej Wąsowski
Shiyi Wei
Harry Xu
Francesco Zappa Nardelli

29th European Conference on Object-Oriented Programming.
Editor: John Tang Boyland

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Object-Oriented Programming without Inheritance

Bjarne Stroustrup

Morgan Stanley
New York, USA
`bjarne@stroustrup.com`

Abstract

Object-oriented programming is often characterized as encapsulation plus polymorphism plus inheritance. The original Simula67 demonstrated that we could do without encapsulation and Kristen Nygaard insisted that some OOP could be done without inheritance. I present generic programming as providing encapsulation plus polymorphism. In C++, this view is directly supported by language facilities, such as classes, templates and (only recently) concepts. I show a range of type-and-resource-safe techniques covering a wide range of applications including containers, algebraic concepts, and numerical and non-numerical algorithms.

1998 ACM Subject Classification D.1.5 Object-oriented Programming

Keywords and phrases object orientation, generic programming, polymorphism, concepts, encapsulation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.1

Category Invited Talk

© Bjarne Stroustrup;
licensed under Creative Commons License CC-BY

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 1–1

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Programming in the Large for the Internet of Things

Jong-Deok Choi

Samsung Electronics
Suwon, Korea
`jd11.choi@samsung.com`

Abstract

The term *Internet of Things* (IoT) has generated a lot of buzz in the information technology and consumer electronics industries. In the IoT setting, a large number of physically dispersed devices – such as sensors, actuators and processing units – coordinate to bring useful capabilities to the user. A significant portion of these devices may have rather small computation and storage footprints, but at the same time, they can leverage support from potential enormous computation and storage resources via the cloud. Also, a large set of small footprint devices can serve not just a single logical app or service, but also many independent logical apps or services. This requires a careful separation of computational activities and their associated data within a device, for privacy and security purposes. Application development for the Internet of Things gives a whole new meaning to the term "programming in the large," and some of this is likely to be new to the practitioner. This talk will discuss what the IoT environment means to the practical programmer, and what apps and app ecosystems for IoT might look like. The talk will also discuss the issues and open challenges in software engineering brought on by this new environment, pointing towards new opportunities for researchers in our community.

1998 ACM Subject Classification D.2.6 Integrated environments, D.2.11 Software Architectures, D.4.7 Organization and Design, K.6.3 Software Management, J.7 Computers in Other Systems

Keywords and phrases software development methodologies, software architecture, programming model, software engineering, internet of things

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.2

Category Invited Talk

© Jong-Deok Choi;
licensed under Creative Commons License CC-BY

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 2–2

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Software Verification "Across the Stack"

Alexander J. Summers

ETH Zürich
Zürich, Switzerland
alexander.summers@inf.ethz.ch

Abstract

Producing reliable programs has always been tough, and the complexity and variety of programming tasks just keeps on growing. Fortunately, the growth of computing power has also enabled substantial advances in automated reasoning, particularly the development of SMT solvers and automatic theorem provers. In turn, this has resulted in new research directions for program correctness, with the aim of producing tools which can verify complex properties of software automatically.

Developing useful verification techniques requires balancing a number of competing factors. We want to be as expressive as possible in the program properties we can support – if we can write it down, we'd like to be able to prove it. But to progress beyond pen-and-paper efforts, we also need to tailor our approaches such that they can be implemented or encoded by tools, ideally with both precise and efficient results. To make things harder still, if we hope to produce tools which everyday programmers can benefit from, we also need techniques that require manageable interaction from a user, and give understandable feedback.

In this talk, I'll describe some of the fun experiences I've had working on these kinds of problems, from the design of front-end program logics and reasoning techniques to the development of underlying implementation technology, and the tricky encoding challenges that show up in between.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases software verification, program logic, automatic verifier, program correctness, SMT solvers

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.3

Category Invited Talk

© Alexander J. Summers;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 3–3

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Towards Practical Gradual Typing*

Asumu Takikawa[1], Daniel Feltey[1], Earl Dean[2], Matthew Flatt[3], Robert Bruce Findler[4], Sam Tobin-Hochstadt[2], and Matthias Felleisen[1]

1 **Northeastern University**
Boston, Massachusetts, USA
`{asumu,dfeltey,matthias}@ccs.neu.edu`
2 **Indiana University Bloomington**
Indiana, USA
`{samth,edean}@cs.indiana.edu`
3 **University of Utah**
Salt Lake City, Utah, USA
`mflatt@cs.utah.edu`
4 **Northwestern University**
Evanston, Illinois, USA
`robby@eecs.northwestern.edu`

Abstract

Over the past 20 years, programmers have embraced dynamically-typed programming languages. By now, they have also come to realize that programs in these languages lack reliable type information for software engineering purposes. Gradual typing addresses this problem; it empowers programmers to annotate an existing system with sound type information on a piecemeal basis. This paper presents an implementation of a gradual type system for a full-featured class-based language as well as a novel performance evaluation framework for gradual typing.

1998 ACM Subject Classification D.3 Programming Languages

Keywords and phrases Gradual typing, object-oriented programming, performance evaluation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.4

1 Gradual Typing for Classes

Gradual type systems allow programmers to add type information to software systems in dynamically typed languages on an incremental basis [39, 48]. The ethos of gradual typing takes for granted that programmers choose dynamic languages for creating software, but also that for many software engineering tasks, having reliable type information is an advantage. The landscape of gradual typing includes many theoretical designs [26, 29, 39, 40, 46, 53], some research implementations cite3,20,49,52,55, and, recently, the first industrial systems (Typescript [51], Hack[1], Flow[2]).

Despite these numerous efforts, no existing project deals with the full power of object-oriented programming in untyped languages, e.g., JavaScript, Python, Racket, Ruby, or

* Due to a conflict of interest, we could not submit an official artifact for consideration to the ECOOP Artifact Evaluation Committee. However, we have prepared an unofficial artifact that is available at the following URL: `http://www.ccs.neu.edu/home/asumu/artifacts/ecoop-2015.tar.bz2`.

[1] See `http://hacklang.org/` and Verlaguet, *Commercial Users of Functional Programming*, Boston, MA 2013.

[2] See `http://flowtype.org`.

© Asumu Takikawa, Daniel Feltey, Eeal Dean, Matthew Flatt, Robert Bruce Findler, Sam Tobin-Hochstadt, and Matthias Felleisen;
licensed under Creative Commons License CC-BY

```
(define C1
  (class object% (super-new)
    (define/public (m) "c1")))
(define C2
  (class object% (super-new)
    (define/public (m) "c2")))
; f is a mixin, result inherits from C
(define (f C)
  (class C (super-new)
    (define/public (n)
      (send this m))))
(define-values (C1* C2*)
  (values (f C1) (f C2)))
```

```
class C1(object):
  def __init__(self): pass
  def m(self): return "c1"
class C2(object):
  def __init__(self): pass
  def m(self): return "c2"
# f is a mixin, result inherits from C
def f(C):
  class Sub(C):
    def __init__(self): pass
    def n(self): return self.m()
  return Sub
c1cls, c2cls = f(C1), f(C2)
```

Figure 1 First-class classes in both Racket (left) and Python (right).

Smalltalk. Among other things, such languages come with reflective operations on classes or classes as first-class run-time values. See figure 1 for an abstracted example of functions operating on classes.[3] While users of dynamically-typed languages embrace this flexibility and develop matching programming idioms, these linguistic features challenge the designers of gradual type systems. Only highly experimental languages in the statically typed world [34] support such operations on classes, and only our previous theoretical design [46] deals with the problem of how to turn such a type system into a sound gradual type system.

This paper presents the first *implementation* of a sound gradual type system for a higher-order, class-based, practical OO language that supports runtime class composition.? Abstractly, it makes three concrete contributions: (1) design principles for gradual typing in a higher-order, object-oriented context, (2) a new mechanism to make the soundness-ensuring contract system performant, and (3) a novel performance analysis framework for gradual type systems. Our project is based on the Racket language [18], because it comes with a typical untyped class system [16] and a gradual type? system [49] implemented as a library [50]. Furthermore, our previous theoretical design [46] was made with Typed Racket in mind.?

Section 2 describes the design and its base principles, while sections 3 and 4 explain the evaluation. Section 5 presents related work; Section 6 suggests general lessons.

2 The Design Space

The design of object-oriented Typed Racket (TR) is informed by basic principles and a formative[4] evaluation. The first subsection explains the principles. The second illustrates them with a series of examples, focusing on how TR accommodates existing untyped idioms. The remainder sketches the challenges, solutions, and limitations.

2.1 By Principle

The design principles of TR fall into two categories: those derived from our previous work on a gradually typed calculus of first-class classes [46] and our practical implementation

[3] We thank Laurence Tratt for inspiring the Python version.
[4] We borrow the terms “formative evaluation” and “summative evaluation” from the education community [36]. A *formative evaluation* informs the design process; a *summative evaluation* assesses its outcome.

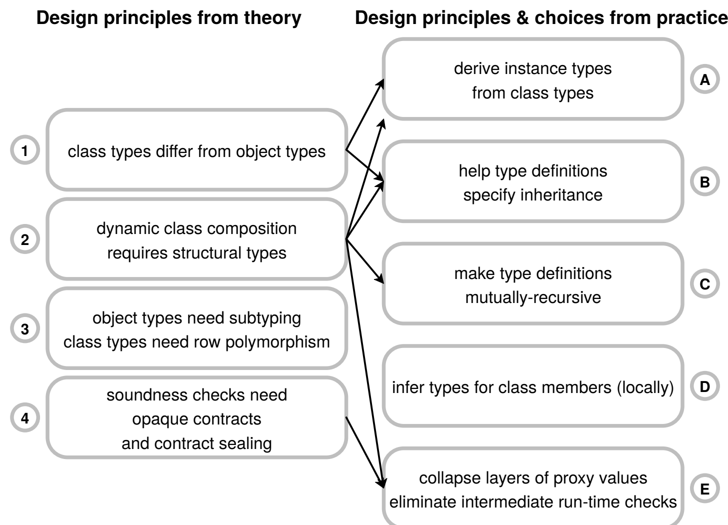


Figure 2 Design principles for Typed OO Racket.

experiences. Figure 2 presents these principles in two columns that correspond to the two categories: the theoretical principles on the left and their induced implementation concerns on the right.

Theoretical principles (1–4). Racket supports classes as first-class values, giving rise to design patterns such as mixins and traits [16]. Class expressions produce class values that freely flow through the program, including functions and data structures:

```
; several class values used at run-time
(define (make-class base-class)
  (class base-class
    (super-new)
    (define/public (new-method arg) ....)))
(define c% (make-class object%))
(list c% c% c%)
```

The first line shows a function that maps a class to a subclass. Such functions are *mixins* because they add new points to an existing class hierarchy. The second line shows a use of the mixin, the third one a list of three class values.

Since class values may appear anywhere and without a statically-known name, we cannot simply identify classes by their names as in *nominal type systems* such as those of Java or C#. Instead we introduce class types that are compared *structurally*. Put differently, classes are not identified just by a name, but by the types of their members (i.e., methods, fields, etc.). This matches up with principle (2) in figure 2; it is also unsurprising as other researchers have proposed a similar approach [3, 5, 8, 33, 52].

Furthermore, the type system must distinguish the types of classes from the types of objects, because expressions can yield either of these; see principle 1. In addition, class types must support polymorphism because mixins are parametrized over base classes. To accommodate this idiom, TR's type system borrows *row polymorphism* from the extensible record and object literature; see principle 3.

By (3), the type system restricts row polymorphism to class types and disallows *width subtyping*; allowing both is unsound [46]. Conversely, it accommodates existing design patterns via object types that have width subtyping but lack row polymorphism.

Unlike an ordinary type system, a *gradual* type system must also support the sound interoperation of typed and untyped code. In TR, *higher-order contracts* mediate these interactions. Concretely, when a typed module in a gradually-typed language exports to/imports from an untyped one, the type system must translate any type specifications into contracts that uphold the type's invariants at run-time. On top of ordinary class contracts [44], principle (4) requires *sealing* contracts for protecting mixins and *opaque* contracts that prevent untyped programs from accessing hidden methods on objects. These contracts are applied to the actual class values that flow to untyped code, and thus the untyped code always interacts with typed values through a protective wrapper that identifies type violations by *blaming* the responsible untyped code.

Practical principles (A–E). Principles based on a small calculus [46] never suffice for a real-world design. In the process of creating TR's gradual type system, five additional design concepts emerged, which we consider as fundamental as the theoretical ones.

The separation of class types from object types calls for a linguistic mechanism that derives types of instances from types of classes or vice versa. The key to this syntax design is to choose a convenient default choice. From this perspective, it is important that instance types are easily synthesizeable from a class type while the type of an instance lacks information – e.g., the constructor – that is needed to reconstruct the type of the class. This insight naturally leads to a choice that makes class types the primary vehicle and introduces an `Instance` type constructor for the derivation of object types from class types.

While a gradual type system for dynamic classes demands the introduction of structural types, writing down such types imposes a significant notational overhead. In practice, class definitions consist of nests of deeply interwoven "has-a" or "is-a" relationships. For example, one class type from TR's standard GUI library consists of 245 methods signatures, and moreover, the type refers to instances of 23 other class types plus itself. Not surprisingly, some of these 23 class types refer back to the original class type. In short, a gradual typing system for a dynamic object-oriented language needs a powerful construct for defining large mutually recursive types.

To accommodate sharing of features in class types, TR's `Class` type constructor comes with an optional `#:implements` clause that allows a given class type to copy another class type's member definitions. Principle (B) captures this point.

One way to accommodate nests of self-recursive and mutually-recursive type definitions is to encode them with ordinary μ-types [11], which already exist in functional TR. Simple experiments in writing types for our GUI library expose the drawbacks of this approach. Therefore TR instead provides `define-type`, a novel[5] form of mutually-recursive structural type definitions. It makes up for the lack of type recursion through class names that nominally-typed language such as Java get for free. These named class types are *not* nominal as in Java because the equivalence between two type definitions is always determined by structural comparison and recursive unfolding. At the same time, our type definitions retain the simple syntax of type aliasing.

[5] Superficially, these mutually recursive type definitions resemble OCaml's mutually recursive classes, but the two differ starkly. Our design separates the class type definitions from the definitions of the actual classes and allows semantic forward declarations without imposing syntactic ones.

Real-world programming also means reducing the number of *required* type annotations. A gradual type system therefore needs an algorithm that reconstructs some types automatically (principle D). Module-level type inference à la Hindley & Milner causes too many problems, however. Hence, TR *locally* infers the types of class members when possible, e.g., fields initialized with literal constants.

Finally, by its very existence, the *theoretical* work [21, 25, 41] on collapsing higher-order run-time checks implicitly conjectures that layering proxy wrappers around values causes a degradation in the performance of gradually typed programs. In contrast, the extensive use of *functional* Typed Racket over six years in open-source and industrial projects had up to now not provided *practical* evidence for this prediction. The addition of object-oriented features to Typed Racket reveals that the possibility is indeed real. The naïve extension of Typed Racket suffers from exponential growth in object wrappers on real-world examples. It is likely that higher-order object-oriented programs are prone to these kinds of interactions due to the widely used model-view architecture. In any case, our work finally demonstrates the practical need of optimizing for space usage.

Constructively put, we articulate principle (E). All implementations of sound gradual type systems with structurally typed classes need a way to merge layers of proxy wrappers. One such implementation in the literature, Reticulated Python [52], strictly follows Siek and Wadler's theoretical proposal of collapsing casts into "threesomes" [41]. Typed Racket employs an alternative solution, which is sketched in section 4.3.

2.2 By Example

To provide evidence that our design ideas from the previous section scale to real code, we walk through a series of examples extracted from our case studies described in section 3. In particular, the examples demonstrate how to add types to untyped code, how typed and untyped code interact, and how the type and contract systems can handle untyped idioms.

Augmenting an existing codebase. Recall that the gradual-typing thesis states that a maintenance programmer ought to be able to augment an existing untyped code base with types in order to benefit from their software engineering advantages, and to ensure that future programmers will continue to receive those benefits. Our illustration starts with an excerpt from the Racket Esquire program in figure 3 [15], which implements a bare-bones interactive editor combined with a Lisp-style Read-Eval-Print-Loop. The excerpt showcases the definition of a typical mixin.

This particular mixin, named `esq-mixin`, adds REPL capabilities to a base text editing class, such as the `text%` class from the GUI standard library on the last line.

The body of the mixin uses `class` to derive a subclass from `base-class`, the function's parameter. The rest of the class form contains typical elements of object-oriented programming: a call to a superclass constructor, several private fields, public method definitions, and an overriding method definition. As in C#, overriding methods in Racket are explicitly signaled with a `define/override` keyword. The `inherit` form both ensures that the superclass contains the given method names and allows the given superclass methods to be called without explicit `super` calls. The call to `new-prompt` in the class body executes when an instance of the class is constructed. In general, any expressions in the class body are run as part of the class's instantiation process.

Our code snippet in figure 3 calls for two pieces of type information: the mixin's argument and the public methods in the mixin's result. Furthermore, since the mixin is polymorphic,

```
(define (esq-mixin base-class)
  (class base-class
    (super-new) ; run the superclass initialization
    ; inherit methods for use from the superclass
    (inherit insert last-position get-text erase)
    (define prompt-pos 0) (define locked? #t) ; private fields
    (define/public (new-prompt)
      (queue-output (λ () (set! locked? #f)
                          (insert "> ")
                          (set! prompt-pos (last-position)))))
    (define/public (output str)
      (queue-output (λ () (let ([was-locked? locked?])
                            (set! locked? #f)
                            (insert str)
                            (set! locked? was-locked?)))))
    (define/public (reset)
      (set! locked? #f) (set! prompt-pos 0)
      (erase)
      (new-prompt))
    (define/override (on-char c)
      (super on-char c)
      (when (and (equal? (send c get-key-code) #\return) (not locked?))
        (set! locked? #t)
        (evaluate (get-text prompt-pos (last-position)))))
    (new-prompt))) ; method call during class initialization
(define esq-text% (esq-mixin text%)) ; application of the mixin
```

Figure 3 Racket Esquire, untyped.

we explicitly specify a row variable for the mixin's type. Figure 4 shows the mixin with a type annotation that encodes the required information.

The top of figure 4 displays a definition for `Esq-Text%`, the type for `esq-text%`. We also use it for the type annotation on `esq-mixin`. We take advantage of the `Class` constructor's built-in support for type inheritance to make `Esq-Text%` inherit from the `Text%` type defined in TR's base type environment.

Since `esq-mixin` is a row polymorphic function, we annotate it with the `All` and `->` type constructors. Given the two type definitions `Text%` and `Esq-Text%`, we can describe the domain and result types of the function type. Both `Class` types specify two key pieces: the types are row polymorphic due to the `#:row-var r` clause and the types inherit from existing structural types `Text%` and `Esq-Text%`, respectively. The former indicates that the class contains an unspecified set of additional methods or fields that are determined when the mixin is actually applied.

The method types inherited from `Text%` and `Esq-Text%` are used to type-check the body of the mixin. For example, the types of inherited methods such as `insert` are deduced from `Text%`. The types on the public methods are given in the definition of `Esq-Text%`. The types for the public methods document the methods' arguments and their effectful nature.

In this example, Typed Racket requires a single annotation to type-check the mixin, which reflects principle E from figure 2 about reducing the burden on the maintenance programmer. In particular, none of the private members or inherited fields need annotations.

```
; in module Library
(define-type Esq-Text%
  (Class #:implements Text%
         [new-prompt (-> Void)]
         [output (String -> Void)]
         [reset (-> Void)]))

(: esq-mixin (All (r #:row)
               (-> (Class #:row-var r #:implements Text%)
                   (Class #:row-var r #:implements Esq-Text%))))
(define (esq-mixin base-class) (class base-class ....)) ; as before
```

Figure 4 Racket Esquire, typed.

```
(Class #:implements Text%             (class/c ; many cases elided from Text%
       [new-prompt (-> Void)]                  [new-prompt (-> void?)]
       [output (String -> Void)]               [output (-> string? void?)]
       [reset (-> Void)])                      [reset (-> void?)])
```

Figure 5 Translating a type to a contract.

Cooperating with untyped code. Imagine that `esq-text%` is integrated into a complete project where other code – say an IDE system – remains untyped. In order to ensure that the widget's type invariants are upheld, the system must dynamically check that the untyped code uses the widget safely.

Concretely, consider a program divided into Library and Client modules. The Library module provides the Esquire text editing functionality, while the Client module uses it as part of a larger program. Since the Client imports the `esq-text%` class, the class value itself flows from the Library, passing through a boundary between the typed world and untyped world on the way.

Now consider a concrete snippet from the Client module:

```
; in module Client
(require "library.rkt")
(define repl-text (new esq-text%))
(send repl-text output 42)
```

The method invocation with **send** clearly violates the `String` type specified on the `output` method. Although the type-checker would catch such a mistake for the Library module, it is unable to inspect the untyped Client code.

Instead, the type-checker translates the type to a contract that ensures the type invariants. Figure 5 shows the result of (automatically) translating the type for the Esquire class to a matching class contract. As explained in section 2.1, these class contracts are opaque, meaning they disallow the export of a class with methods not explicitly listed in the type.

When classes flow from untyped modules to typed modules, the typed code must specify types for these imports. Suppose that we need to import the `text%` class directly from the standard, untyped GUI library. Assuming `Text%` is defined, the typed portion could use an import specification like this:

```
(require/typed racket/gui
               [text% Text%])
```

The `require/typed` form in TR imports the given bindings with the given type specifications. As before, these types are translated to contracts, ensuring that the imports live up to the desired specifications.

Mixins and typed-untyped interoperation. Let us illustrate the typed-untyped interoperation with an example of a mixin from the Big-Bang event-based functional I/O library. The library uses mixin methods such as these:

```
; if no callbacks are provided (on-key, on-pad, on-release), don't mix in
(define/public (deal-with-key base-class)
  (if (and (not on-key) (not on-pad) (not on-release))
      base-class
      (class base-class
        (super-new)
        ; the method invokes the callbacks supplied by the user
        (define/override (on-char e) ...))))
```

This method accepts a class argument named `base-class` and, when appropriate, returns a subclass that adds a custom key event handler. The method's implicit precondition requires that the class `base-class` already contains a `on-char` method to be overridden. The remaining members of the class are unconstrained.

To import a class with such method, a programmer may write

```
(require/typed
 [world% (Class ...
                [deal-with-key
                 (All (r #:row) ; method types elided for space
                   (-> (Class #:row-var r [on-char ...])
                       (Class #:row-var r [on-char ...])))])])
```

The (automatic) translation of a row-polymorphic function type into a contract requires a seal [46] for the `deal-with-key` method:

```
(sealing->/c (X) [on-char]
   (and/c X (class/c [on-char ...]))
   (and/c X (class/c [on-char ...])))
```

The `sealing->/c` combinator creates a function contract that generates a fresh class seal when the wrapped function is applied. The occurrences of `X` in the body of the combinator are replaced at run-time with either a sealing or unsealing operation depending on whether the variable occurs in a negative or positive position. Each seal lists those class members that are left unsealed (here, the `on-char` method); all unmentioned members are hidden. Thus the method contract above explicitly seals off all members from the argument class `base-class` in the implementation except `on-char`. This prevents the mixin from adding or overriding any methods other than `on-char`, which matches the polymorphic type and expresses the intent of the method in a precise manner.

Examples of mutually-recursive type definitions. The complex relationships between classes in practice requires the use of mutually-recursive type definitions, see figure 2 (C). The type for the `text%` class is highly illustrative:

```
(define-type Text-Object (Instance Text%))
(define-type Text%
  (Class ; 244 other methods ...
         [set-clickback
          (Natural Natural (Text-Object Natural Natural -> Any)
           -> Void)]))
```

The `set-clickback` method for `text%` objects installs a callback triggered on mouse clicks for a region of the text buffer. The type for that callback function recursively refers to the `Text%` type via the `Text-Object` definition. No explicit recursive type constructors are necessary to write the type down. Under the covers, TR establishes the recursion through the type environment, even though `Text%` is not a nominal type.

2.3 From Types to Contracts: An Implementation Challenge

Soundness calls for run-time checks that enforce the type specifications when a value flows from the typed portion of a program to an untyped one. All theoretical designs choose either casts or contracts for this purpose. GradualTalk [3] and Reticulated Python [52] rely on the former; research in this realm focuses on what kind of casts to implement and how this choice affects the expressiveness of the language and the efficiency of programs. In contrast, TR is the first implementation of a gradually typed, object-oriented languages that uses higher-order contracts instead of casts.

Although the homomorphic translation from types to contracts (illustrated above) is easy to use in theory work [46], implementing it for practical purposes is not easy. For the kinds of structural types used in the TR code base, it generates excessively large contracts. To achieve reasonable results, it is critical to treat the problem of translating types to contracts as a compilation problem that requires an optimization phase.

Based on several failed attempts, TR now compiles a recursive type to a recursive declaration of mutually recursive contracts for each dependency. To create contracts of manageable size, the analysis of the dependencies among the type definitions finds cycles in the dependency graph. Definitions within these cycles are lifted and memoized. Definitions that do not participate in cycles incur no overhead.

2.4 Limitations

TR currently suffers from a few limitations, which include some workarounds for our case studies, but rarely prevent the conversion of an untyped module.

Row polymorphism on objects. Our system provides row polymorphic types for supporting mixins, but only class types are allowed to contain row variables. Unlike most designs with row polymorphism such as OCaml, the types for objects are concrete and support standard width subtyping. This tradeoff works well for most of our examples, because Racket programmers are often content with Java-style use of classes. Our choice rules out row polymorphic functions that construct an object from a given class or the use of row polymorphism to emulate bounded polymorphism for objects. In practice, the lack of row polymorphism for objects prevents us from porting a *single* module in the DrRacket IDE. We conjecture that adding bounded polymorphism to Typed Racket would fill this gap.

Occurrence typing for OO code. One of the important features of functional TR is its use of *occurrence typing* [49]. To accommodate dynamic type-tag tests, TR refines the types

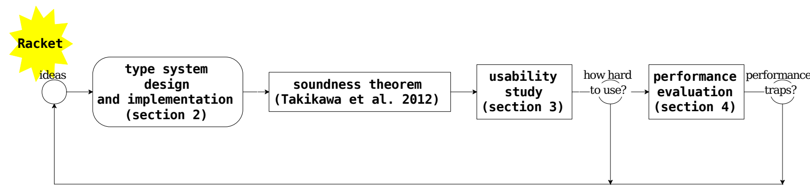


Figure 6 The feedback cycle for Typed OO Racket.

of variables depending on where they occur. If, say, a dynamic test checks whether `x` is a non-empty list or `y` is a number in a specific interval, their types in the `then` and `else` branches of a conditional reflect the possible results of these checks. Occurrence typing is crucial for porting untyped programs into the typed world because the former often discriminates elements from unions of data via predicates.

Sadly, while TR supports occurrence typing on private class fields, it cannot support two important uses for OO constructs: (1) recovering object types from uses of `is-a?`, which is like `instanceof` in Java, and (2) occurrence typing on public fields. For an example of the first, suppose we export the typed `esq-text%` class from earlier to untyped code. If we encounter the test `(is-a? an-object esq-text%)` in typed code, we would like to conclude that the value `an-object` has the type `(Instance Esq-Text%)`. Unfortunately, this is unsound if `an-object` originates in untyped code, since the untyped code may have subclassed `esq-text%`, overridden its methods with ill-typed implementations, and constructed `an-object` from that subclass. Closing this gap requires additional research.

For the second problem, consider how a concurrent thread may mutate a public field between a tag check and the execution of the corresponding branch of the conditional. An application of occurrence typing could then lead to an incorrect type for a field based on an out-of-date tag check. We therefore will investigate immutable public fields in the future.

3 Effectiveness Evaluation

Like all good design efforts, our design of Typed Racket takes place in the context of a feedback cycle. Figure 6 visualizes our particular feedback loop. With respect to this paper, two elements stand out: the usability study and the performance evaluation. This section presents the former for Typed OO Racket and its design impact. The next section introduces a novel performance evaluation framework and discusses how it influenced the design.

The usability evaluation aims to test three hypotheses:[6]

1. Typed Racket enables programmers to add types in an incremental manner, including for components that dynamically create and compose classes. This hypothesis demands two specific qualities from the type annotation process: the burden of adding type annotations must be small, and the program logic should rarely change.
2. Each theoretical design principle (1–4) is needed for realistic programs.
3. Each practical design principle (A–E) helps annotate realistic programs.

[6] Our goal is *not* to determine whether static typing per se contributes to software maintenance, deferring instead to the existing literature [22].

3.1 Cases

The evaluation was conducted in two stages: a formative evaluation and a summative one. For the summative evaluation, the programmer had no prior knowledge of Racket's class system or Typed Racket's gradual type system.

With two exceptions, the code bases in this section come from the mature Racket distribution. The newest dates from Racket version 5.3.2, released in January 2013; the rest have been shipped in user-tested distributions for a minimum of two years and some for nearly twenty years. The two exceptions are Acquire and Esquire.

Our *formative* evaluation employed four case studies: a tool that inserts Large Letters into a program text, a Tooltip drawing library, the functional Big-Bang I/O library [12], and Esquire. The Big-Bang case study is about annotating the library's graphical core while leaving the remaining pieces untyped. Finally, Esquire is a graphical REPL that illustrates the essential elements of DrRacket; some of the code snippets in section 2 are taken from Esquire.

The nine cases included in the *summative* evaluation cover the full range of object-oriented programming idioms in Racket. Several cases are extracted from a package of games that are included in the Racket distribution: Mines, a graphical Minesweeper game, Slidey, a puzzle game, Cards, a library for the standard 52-card deck, and Aces and GoFish, two games using Cards. The Markdown component is one of several renderers for the Scribble documentation language [14]. The DB case study covers a library that provides access to SQLite databases. Finally, the Acquire board game is a project from a programming course that represents an interactive system with a user API.

Since the purpose of the case study is to evaluate a *gradually* typed system, the addition of type annotations to modules was not exhaustive. Instead, the key modules of each program were ported, with an emphasis on modules that used objects and classes.

3.2 The Process

Some of the code comes with comments, behavioral contracts, or documentation that describe the "types" of methods and fields. Injecting types into such pieces of code is often straightforward, though the specifications are sometimes out-of-sync with the program logic. When the code lacks specifications, the maintainer must reconstruct them from the program logic. This ranges from easy (e.g., for fields with an initial value) to difficult (e.g., methods with complicated invocation protocols).

Over the course of the typing process, a developer iteratively acquires an understanding of the code and adds type annotations until the type-checker is satisfied. In practice, the developer may need to modify the program logic or add assertions or casts to force type checking. Even after the type-checker approves the component, the typing effort is not over. Components that interact with other untyped components do not run correctly if an impedance mismatch exists between the types specified in an import statement and the run-time behavior of the untyped components. Hence, the developer runs the program on its test suite and improves the types in response.

3.3 Quantitative Results

Concerning metrics, we follow the precedent for functional Typed Racket [47]. These metrics are chosen to judge whether a developer can gradually equip a code base with types: the size of the code plus the number of type declarations, type annotations, and type assertions. The latter are important because the annotations are also software artifacts for which a developer must accept maintenance responsibility. In addition, we report how many changes

Program	Let	TT	BB	Esq	Mi	Card	Mdn	DB	Acq	GF	Ace	Sld
Lines	216	218	1077	177	533	620	328	749	1419	443	333	357
% Increase	2	7	11	11	13	19	16	23	20	11	5	15
Useful ann.	11	9	85	8	22	38	27	31	83	30	13	21
λ: ann.	0	0	15	4	38	5	4	3	19	12	10	2
Other ann.	14	7	29	0	4	17	0	2	12	5	0	4
Type def.	0	0	7	0	6	5	1	13	21	1	2	2
Typed req.	0	0	20	0	0	1	3	35	71	3	2	0
Assert/cast	4	3	25	1	10	4	11	5	13	3	0	9
Ann./100L	12	8	12	7	13	10	10	5	9	11	8	8
Problems	4	3	12	1	5	5	2	6	4	1	2	1
Fixes	1	1	1	0	1	1	2	2	0	1	0	1
Theo. princ.	4	1–4	1–4	–	1,3	1,3,4	2–4	1–4	1,3	–	–	–
Prac. princ.	D	A,D	A,B D	D	A,B D	A,B D	B,D	A–D	A–E	A	A	A,B D
Time	–	–	–	–	9h	7h	7h	7h	11h	1h	4h	5h
Difficulty	⋆	⋆	⋆⋆⋆	⋆	⋆	⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆	⋆	⋆

Figure 7 Case study results.

to the program logic are needed to accommodate the type system, because such changes may potentially alter the program's behavior.

Figure 7 reports the results in a table. The detailed interpretations of the rows are as follows. The *Lines* row indicates the total number of lines in the ported program while the *% Increase* row denotes the percentage of the total added by porting. The *Useful annotations* row consists of the number of identifiers given types; for example, the method type in the following excerpt from Big-Bang counts as useful:

```
(: show : Image -> Void)
(define/public (show pict0) ...)
```

In contrast, annotations added for the sake of the type-checker are not counted in this category. The *λ: annotations* row contains the number of annotations for function parameters of typed lambda expressions; these do not count toward the useful category because their types are often obvious from context, but the type system cannot infer them. The remaining *Other annotations* category covers the rest.

The *Type definition* row describes the number of type definitions added. As section 2 explains, many uses of `define-type` introduce names for class types. The *Typed require* row counts the number of bindings that are imported from untyped code using `require/typed`. The *Assertions / casts* row counts the number of assertions and casts used to assure the type-checker. The *Ann. / 100L* row shows the number of type annotations per one-hundred lines of code, rounded to the nearest integer.

The *Fixes* row indicates the number of error (correction)s due to types while *Problems* counts the changes made to circumvent limitations or over-approximations in the type-checker.

The *Theoretical/Practical principles* rows note principles (1-4) and (A-E) from figure 2 that apply to the code base. The *Time taken* row measures the number of hours (rounded up) taken to annotate and modify the code and finally the *Difficulty* describes the subjective difficulty of porting the code from ⋆ (easy) to ⋆⋆⋆ (hard).

All case studies rely on types for Racket's standard libraries, i.e., the base type environment. In addition to the core bindings provided by functional Typed Racket, our case studies use extra standard libraries such as the GUI libraries, drawing libraries, and core documentation

libraries. We do not count the annotations in the base type environment for the line numbers above because they are shared across all programs.

3.4 Qualitative Results

Principles. Figure 7 lays out which design principles from figure 2 were necessary for porting each code base. While subjective, we judged each code base with consistent criteria for each principle. For example, we decided that a code base required principle (1) if it used both class and object values. If a code base additionally used `Instance` type constructors, it fit principle (A). Principles (2) and (3) applied to any code base that used row polymorphism. Additionally, we checked (2) for any code base that used obviously structural types.

For (B) and (C), we determined whether the code base *directly* used `define-type` with a `#:implements` clause or with mutual-recursion, respectively. All of the case studies except Markdown, DB, and Acquire used mutually-recursive type definitions because of their dependence on the GUI standard library, but we did not count these indirect uses. For (4), we included any code bases that exported classes and/or objects to untyped code or imported them from untyped code. We recorded (D) if classes in the code base left out type annotations that TR would reconstruct. Only Acquire needed (E) due to exponential proxy layering.

Difficulty. The projects marked with ⋆⋆⋆ in the case studies share some of the following characteristics: (1) the data definitions and the code structure pose comprehension problems, (2) the code base uses language constructs that are difficult to describe with types, or (3) the control flow of the program makes the synthesis of type annotations difficult. The Markdown program fits the first case due to the use of 29 nested and recursive data structures in its logic. The Big-Bang program exemplifies the second characteristic. In Big-Bang, the primary class uses methods that act as mixins on other class values. Furthermore, the program also uses synchronization constructs for concurrency, syntactic extensions that construct methods, and I/O through the graphics layer and the networking library. Finally, the DB library uses complicated error handling that requires the programmer to track control flow when adding type annotations.

3.5 Problems and Fixes

The case studies identify several pain points in the system. Broadly speaking, these points take the form of syntactic overhead in type annotations or additional code necessitated by the type-checker. Here we list the three worst problems and explain how TR addresses them.

First, the `#:implements` shorthand for writing class types does not copy the types for constructor argument types because the constructor arguments may change. That is, a subclass does not necessarily use a superset of its parent constructor arguments. This design decision incurs some cost – in the form of larger types – for the case studies. To eliminate this limitation, TR now comes with a `#:implements/inits` form, which propagates the constructor types.

Second, the lack of occurrence typing for public fields forces a workaround to satisfy the type-checker. The workaround declares a local variable that holds the current value of the public field, which enables type refinement via occurrence typing on the local version. This works only when the field is not modified concurrently.

Third, the type system cannot propagate occurrence typing information or reason with enough precision about the expansion of syntactic extensions. These cases require re-writes with different functions or syntactic forms. An example of the former occurs in the Big-Bang

library, for which the excerpt `(inexact->exact (floor x))` is rewritten to `(exact-floor x)` to accommodate the type-checker. For the latter, our port of DB modifies a use of the `case` form, which provides simple pattern matching, to use `cond`, a general conditional form. Fortunately, both rewrites are simple and local.

3.6 Discussion

For the first hypothesis, we consider whether the effort of adding types to the code bases is reasonable. The overall increase in the number of lines for our object-oriented programs – about 15% across all code bases – is greater than the 7% increase across all of the programs ported in the functional world [47]. We conjecture two explanations: (1) structural type specifications for object-oriented programs are often larger and more complex than function types, and (2) the object-oriented part of Typed Racket lacks syntactic support for formulating concise types. In particular, Typed Racket does not derive class types from class definitions.

Additionally, the code bases that we could *not* include in our investigation are relevant for the first hypothesis. We rejected several code bases from the case study for three reasons: they use Racket's *first-class* modules, they call for the reflection API on records, or they require row polymorphism for objects. Only the third omission points to a flaw in our type system design; the first two are general Typed Racket limitations. The third point prevented the porting of only a single small module, and we thus consider the first hypothesis validated.

As for the second hypothesis, seven out of the twelve case studies require half or more of the theoretical design principles from figure 2. The ones that require the fewest are Esquire, Go Fish, Aces, and Slidey, which are all self-contained and do not use mixins. Due to their self-contained nature, these programs also do not require extensive contract checking.

For the third hypothesis, we also see that most of the case studies rely on three or more of the practical design principles. The least directly used is mutually-recursive type definitions (C). As noted above, however, the feature is heavily employed in the common base type environment and is therefore indirectly used everywhere.

4 Performance Evaluation

Gradual typing suggests that a programmer who performs maintenance work on existing code (re)discovers the types behind the design and adds them to the code base. Each conversion may put up a new higher-order contract boundary between two components, which may have negative consequences for the performance of the overall system. In theory, a completely converted program – without casts or type assertions – should have the same performance as a completely untyped one, because the type checker simply removes types in this case.[7]

A performance evaluation must therefore consider all possible paths from an untyped system to a completely typed one through the lattice of possibilities. Thus far, the gradual typing literature has not presented any results for such experiments. In this section we present the first results of such a performance analysis and explain its role as a formative element of our design process. Specifically, we explain our methodology in some detail, present the results of two experiments, and explain the performance pitfalls that these formative evaluations found and our fixes.

[7] Typed Racket currently performs local optimization on some simple datatypes [50].

4.1 Methodology

The runtime cost of a (sound) gradual type system is a function of boundaries between typed and untyped pieces of code. As the programmer chooses to create such boundaries via type annotations, the system migrates through a space of possible boundary configurations. In Typed Racket, these boundary configurations are determined mostly by which modules are typed and untyped. The path through the space of mixes of typed and untyped modules starts at the fully untyped program. At each step, one more piece is annotated with types.

For example, the lattice in figure 8 shows the simplified configuration space (of four modules) for Acquire. Each node in the lattice contains four boxes whose shading indicates whether it is typed or untyped. The horizontal bottom box represents an I/O library that remains untyped throughout the process.[8]

During the implementation of Typed Racket, we used the following, modest working hypothesis based on Safe TypeScript's experience of 72x slowdowns [32, p. 12]:

> *no path through the lattice of configurations imposes an order-of-magnitude degradation of run-time performance at any stage.*

Pragmatically put, a programmer may add types anywhere in the program without slowing down the program more than a factor of 10.

We used two of our case studies to investigate this hypothesis: Acquire and Go-Fish. For each case study, we converted the primary modules – excluding infrastructure modules such as the I/O library mentioned above – and then scripted the generation of all possible configurations.[9] Each configuration was run 30 times on a stand-alone Linux 3.16 computer with an Intel Core i7-3770K CPU and 32GB of memory.

4.2 Results and Preliminary Interpretation

Figure 8 annotates the lattice of configurations with timings – shown below the modules – that are normalized to the baseline of the fully untyped configuration. The figure also displays normalized standard deviations. An inspection of the lattice reveals that choosing the outermost paths degrades the performance quickly, while some of the innermost paths reduce the performance in a gradual manner. At the worst points, a gradually typed program is almost 40% slower than an untyped program; at the top, we find a fully typed program that is still 39% slower than the fully untyped program.[10]

A large fraction of the additional cost is due to the boundary between the untyped library modules in Acquire (the horizontal white box in the figure) and the typed modules. Some of the cost is due to the boundaries among the four central modules. Critically no configuration slows down performance by an order of magnitude.

Figure 9 collects the results of evaluating the lattice for the four modules[11] in Go-Fish. Many paths from bottom to top go through only one point in which the performance

[8] Adding coarse types to this library is easy but useless; adding precise types is currently impossible.

[9] The scripting required introducing additional typed wrapper modules in some cases to provide extra type signatures. We consider these wrappers to be part of the infrastructure modules.

[10] Earlier versions of Typed Racket caused overheads of up to around 3.4x slowdown on Acquire but still did not exceed 10x. The overhead was brought down to current levels by compiling object types to more efficient contract forms.

[11] We started out with five modules in the Go-Fish experiment but found that one module was never run – and thus did not affect the benchmark runtime – therefore we chose to keep it untyped to make the results clearer.

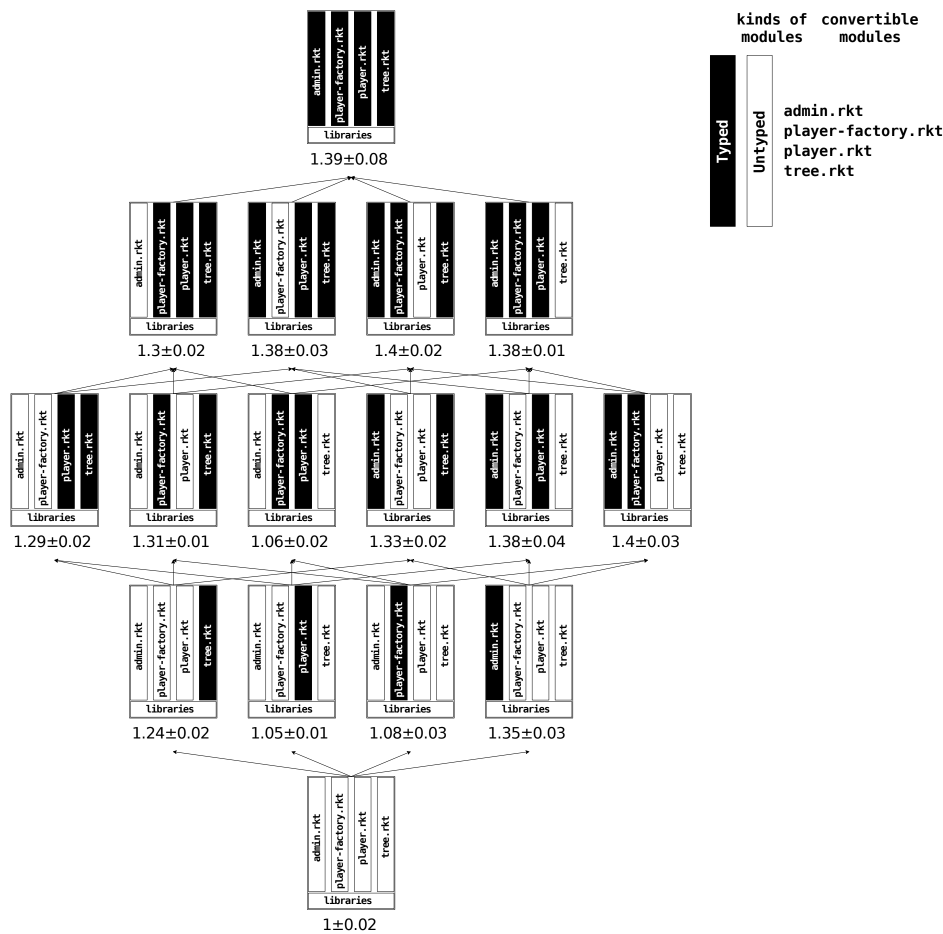


Figure 8 Lattice results for Acquire.

degradation is worse than 10%. There is also a single path along which the overhead never exceeds 5%. In no case does the timing exceed 1.33x of the baseline. The lattice also shows which modules add costs when they are typed. For example, the lattice shows that the typed `classes.rkt` module adds significant overhead unless `gofish.rkt` is also typed.

4.3 Problems and Fixes

The first implementation of OO support for Typed Racket falsifies even the modest working hypothesis, revealing a major problem in the language's run-time infrastructure. Concretely, running some configurations exhibits exponential degradation in performance, both in terms of space and time consumption. Indeed, some configurations no longer terminate. This section explains the problem and sketches how our second implementation of OO support for Typed Racket solves it. This solution is an implementation of principle (E) from figure 2.

As mentioned in section 2.1, Typed Racket compiles types for module exports and imports into Racket's contracts so that untyped modules cannot violate the type invariants. For plain values, a contract just checks that the value conforms to the type. For higher-order values, e.g., objects and classes, the contract system wraps the value with a proxy value – called

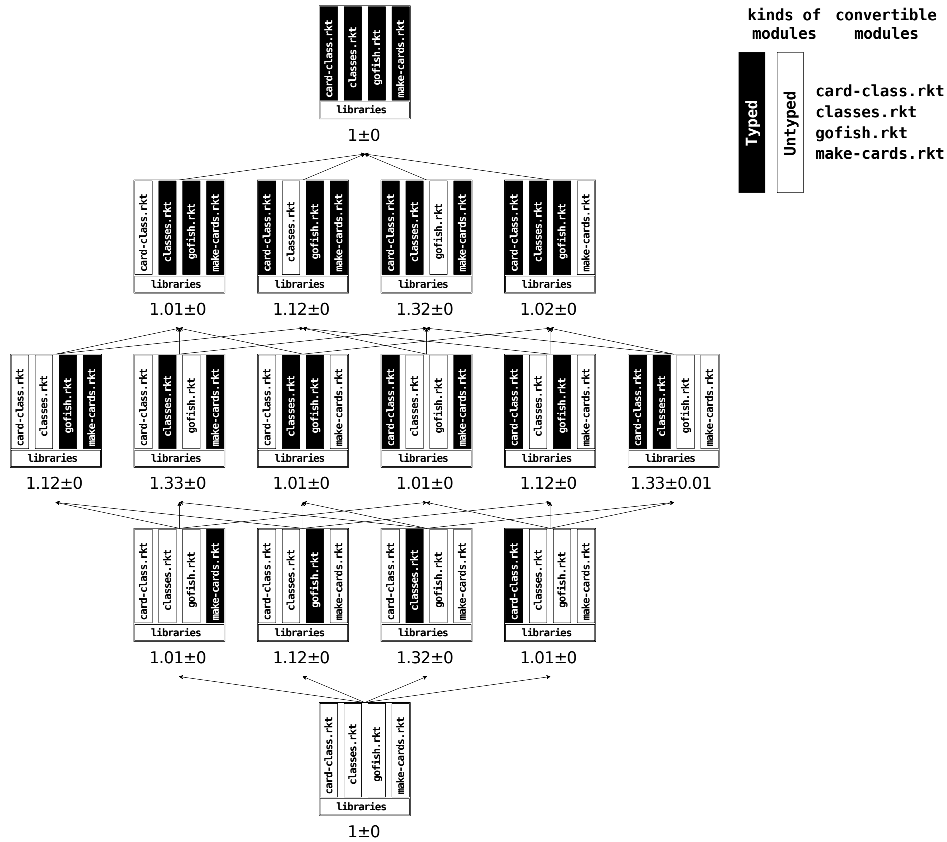


Figure 9 Lattice results for Go-Fish.

chaperones [45] in Racket – that interposes on all accesses and method calls. If the proxy wrapper discovers any bad behavior, it blames the module that agreed to the contract and violated it, which is critical because wrapped values can flow arbitrarily far away from the contract boundary before a violation shows up.

Given this background, the problem is easily explained. Every time a higher-order value flows from a typed module to an untyped one or vice versa, the run-time system wraps it with a proxy layer. A method call may add another layer because the method's signature may impose type invariants. If this boundary-crossing occurs in a loop, layers can quickly accumulate. The growth in layers is exponential with respect to the number of boundary crossings. In the worst case, each round-trip corresponds to a doubling in the number of layers, which explains the exponential consumption of space (wrappers) and time (interposition).

Figure 10 illustrates the idea with a minimal example using a class with a single method; the example distills a problematic two-module fragment from Acquire. Method `m` in `obj` is defined and exported (to `"B.rkt"`) with a recursive, higher-order contract – a common phenomenon in the object-oriented world where classes and objects refer to themselves. The expression `(loop obj)` kicks off the program, starting a loop that calls the method `m` on the object. While this loop uses a finite amount of space in the absence of contracts, the evaluation of `(send obj m obj)` applies the domain contract on `m` to the argument, which is `obj` including its sole method.

```
#lang racket ; A.rkt
(provide (contract-out [obj bubble/c]))

(define bubble/c
  (recursive-contract
   (object/c [m (->m bubble/c bubble/c)])))

(define obj
  (new (class object%
         (super-new)
         (define/public (m x) x))))
```

```
#lang racket ; B.rkt

; The driver module, kicks off the
; problematic loop below

(require "A.rkt")

(define (loop obj)
  (loop (send obj m obj)))

(loop obj)
```

Figure 10 Exponential wrapping.

When the loop starts up, `obj` is already wrapped in one proxy layer due to `bubble/c` from `contract-out`. The domain contract of `m` wraps another layer around `obj`, which increases the number of chaperones to two. Once `m` returns `obj`, the range contract on `m` wraps a final layer around the object, increasing the number to three. For the next iteration, `obj` starts with three layers that enforce `bubble/c`. Since `obj` is *both* the target and the argument in `(send obj m obj)`, each of the domain and range contracts are applied three times, once per existing layer. In short, the number of layers doubles to six. This doubling occurs on each iteration and thus causes exponential memory use.

Our revised implementation of TR solves this problem, loosely based on two theoretical investigations mentioned above. Concretely, Herman et al. [25] compile the contracts in their surface language to a Henglein-style coercion calculus, in which multiple levels of coercions can be eliminated – though without respect for blame information. Siek and Wadler [41] collapse layers of coercions into a representation that includes the greatest lower bound of all types involved in a sequence of wrappers – and thus preserve blame information. Although these theoretical solutions do not apply to Racket's contract system directly, the idea of collapsing layers is applicable to our system.

The current implementation of TR improves the first one in two regards. First, Racket's revised contract system checks whether an existing contract implies one that is about to be wrapped; if so and if the blame information is identical, the new wrapper is not created. Second, Racket's revised proxy mechanism allows dropping a layer in special cases. In particular, some proxies can be marked as containing only metadata with no interposition functions. The run-time system allows such proxy layers to be removed and replaced. By encoding blame information in these metadata proxies, it is possible to replace layers instead of adding redundant ones. This optimization currently works only for object contracts; we intend to generalize it for other language constructs in the future. Together these two changes removed all performance obstacles and enable the revised TR implementation to validate our modest performance hypothesis.

4.4 Threats to Validity

Our formative performance evaluation suffers from some shortcomings that suggest it might not be representative for a summative evaluation. First, the experiments evaluate only the overhead of the contracts created by the typed-untyped boundary. Second, they ignore the overhead due to modifications of the program logic. If a programmer changes the code to

accommodate the type checker or inserts a cast, the current evaluation attributes this cost to the typed/untyped boundaries.[12] Third, our module boundaries may not be representative because we merged some smaller modules (e.g., 10 lines of code) in the code bases into larger ones in order to reduce the lattice size. Since the lattice grows exponentially in the number of modules, exploring the full lattice would take too much time. Fourth, the top of the lattice does not correspond to a program in which every single module is exhaustively typed; infrastructure modules and difficult-to-type modules are left out. Finally, the experiments suffer from somewhat imprecise measurements. In particular, they execute untyped module in a `typed/no-check` mode, meaning the modules still load Typed Racket's run-time library.

5 Related Work

Since this paper reports on the transition from theoretical calculi [4, 40, 46] to full-fledged, gradually typed object-oriented programming languages, this section focuses on implementation efforts of gradual type systems and on prior implementations of typed languages with flexible class composition.

5.1 Gradualtalk

Typed Racket differs from Gradualtalk [3], a gradually typed dialect of Smalltalk, in two major ways. First, TR implements *macro*-level gradual typing using higher-order contracts as the enforcement mechanism at module boundaries. Meanwhile, Gradualtalk uses the *micro*-level approach pioneered by Siek and Taha [39], meaning that Gradualtalk programmers can freely omit type annotations. When they do, Gradualtalk injects the value into the `Dyn` type and downcasts it from there later.

Second, Gradualtalk does not require row polymorphism because Smalltalk projects rarely use dynamic inheritance with mixins or similar features. Classes are declared statically. In contrast, TR necessarily places more emphasis on structural types and row polymorphism to support the numerous dynamic uses of inheritance in Racket.

Due to the differences in the fundamentals, Typed Racket and Gradualtalk's evaluations are necessarily dissimilar. The Gradualtalk evaluation consists of porting an impressive corpus of nearly 19k lines, with the largest typed component consisting of over 9k lines. These Gradualtalk components make significant use of the `Dyn` type, which we conjecture makes porting large numbers of lines easier than in Typed Racket, likely trading type precision. More precisely, for every difficult-to-type phrase, a programmer can use `Dyn` and avoid the hard work of developing a precise type; conversely, replacing uses of `Dyn` may trigger non-local program changes. Qualitatively, the difference in type precision manifests itself at run-time. With `Dyn` types, the dynamic portions may be deeply intertwined with typed portions and thus many more code paths may emit a coercion failure.

In addition, the flavor of ported components differs. Gradualtalk's evaluation includes the Kernel project, which contains the core classes of Smalltalk. Racket's use of classes in the core is limited to those few places where extensibility or GUI hierarchies are needed. Our case studies therefore focus on GUI programs or those, such as Markdown or DB, which are built for extensibility.

[12] The `Go-Fish` experiment runs in headless mode because casts in the GUI code are excessively expensive at the moment. We are investigating the cause.

Concerning performance, the Gradualtalk evaluation does not consider the porting process as a whole. Allende et al. [4] do evaluate the performance of several cast insertion strategies using microbenchmarks.

5.2 Reticulated Python

Like Gradualtalk, Reticulated Python [52] (henceforth Reticulated) implements micro-level gradual typing with `Dyn` types. In an attempt to overcome performance problems, Reticulated implements three styles of cast semantics with different design tradeoffs. The *guarded* semantics is most similar to Typed Racket's use of proxy objects to implement higher-order casts. Unlike the latter, Reticulated uses "threesomes" to avoid repeated proxying. TR does not use "threesomes" because Racket's underlying contract language is more expressive than Reticulated's cast language. Furthermore, the runtime support for contracts (i.e., chaperones) enforces more stringent restrictions than Reticulated's proxies. Like TR, the Reticulated evaluation found that object identity posed a challenge for porting programs in the guarded semantics. The *monotonic* semantics [38] avoids proxying while maintaining blame, at the cost of potential extra errors when interacting with untyped code, but has not yet been fully evaluated in Reticulated.

For recursive type aliases, Reticulated uses a fixpoint computation over its class declarations to determine the recursive object types to substitute into class bodies [52, § 2.1.3]. Meanwhile, TR's **`define-type`** allows the encoding of general mutual recursion between *any* type declarations.

Reticulated's mostly qualitative evaluation does not analyze performance concerns.

5.3 Thorn

Instead of gradually layering typed reasoning on an untyped language, a designer may also choose to embed design elements of untyped languages in a statically-typed language. Notably, the Thorn language takes this approach to support flexible object-oriented programming via *like* types [7, 55]. The uses of a variable annotated with a like type are statically checked, but at run-time any value may flow into such variables. While Thorn's design goals include providing the "flexibility of dynamic languages," its design explicitly leaves out the "most dynamic features" such as dynamic class composition [55]. In contrast, we aim TR specifically at augmenting existing code bases, and thus it necessarily supports the dynamic features that are in use.

In addition, the goals of Typed Racket and Thorn differ in their treatment of blame and when run-time errors may occur. In TR, most run-time checks occur at module boundaries and thus most mismatches are signaled when a module is imported. Thus, if an untyped object imported with a type is missing any specified methods, the contract system immediately blames the untyped module. Thorn, on the other hand, checks method presence only when the object flows into an expression in which the method is used. Thus, a tradeoff is made between flexibility and immediate checking of specifications. Furthermore, Thorn provides no equivalent of blame tracking, trading precision of debugging information for performance.

5.4 Typescript and Hack

Industrial designers of programming languages have started to adopt ideas from the gradual typing research community. In particular, both Typescript and Hack allow programmers to add types to programs in their respective base languages, JavaScript and PHP. These efforts,

like Typed Racket, focus on supporting the idioms in the underlying languages such as traits – an alternative to mixins that emphasize horizontal composition – or prototypes. They make no effort, however, to put the interoperation of typed and untyped code on a sound footing.

Recently, Rastogi et al. [32] proposed Safe TypeScript, which enables safe interoperation for TypeScript. Their approach differs from Typed Racket in using run-time type information for casts whereas TR erases all types after compiling to contracts. Their performance evaluation measures the performance overhead of casts on several Octane benchmarks in two modes, with and without type annotations. Superficially, this is similar to testing TR in both the fully untyped (bottom) and fully typed (top) modes. However, they are not directly comparable because Safe TypeScript incurs a heavy (up to 72x) overhead with no type annotations while untyped Racket code does not incur any overhead until it interacts with a typed module.

5.5 Soft and Strong Type Systems for Dynamic Languages

Type systems that accommodate reasoning for untyped programs have been proposed for many languages. Early work includes soft typing for Scheme [9, 13, 31, 54], polymorphic type inference for Scheme [24] based on the *dynamic typing* [23] formalism, the Strongtalk project [8] for statically-typed Smalltalk with mixins, and Marlow and Wadler [28]'s work on a type system for Erlang. These early proposals do not support interoperation as defined by gradual typing. In soft-typing, run-time checks are inserted where the type system cannot reason with the given rules. The checks come without blame. Strongtalk provides an idiomatic type system for Smalltalk, but offers only "downward compatibility" [8] (i.e., Strongtalk code elaborates to valid Smalltalk). The elaboration is not sound for interoperation with Smalltalk in the sense of gradual typing. Several ideas used in Strongtalk, e.g., "brands" and "protocols", are relevant for future extensions to Typed Racket such as nominal typing and more concise types.

More recently, several designs in this space use a variety of techniques such as type inference, dependent types, and refinement types to support idioms in dynamically-typed programs. DRuby [20] uses type inference to discover types errors in Ruby programs and inserts run-time checks if the programmer supplies type annotations. Dependent JavaScript [10] supports JavaScript idioms found in real world programs through the use of dependent types with a refinement logic, off-loading some of the reasoning to an SMT solver. While these systems are not gradually-typed, their techniques will be helpful for future improvements to gradual typing of objects.

5.6 Types for Mixins and First-class Classes

Our work is inspired by a long line of research on semantics and type systems for mixins and objects. The literature on mixins has focused on class-based languages, many inspired by Java or Smalltalk. In the object world, classes are encoded as syntactic sugar as in the σ-calculus [1] or ML-ART [34].

Many models of mixins or first-class classes have been proposed for Java-like languages: Flatt et al. [17]'s MixinJava, Ancona et al. [6]'s Jam, McDirmid et al. [30]'s Jiazzi, Allen et al. [2]'s MixGen, Kamina and Tamai [27]'s McJava, and Servetto and Zucca [37]'s MetaFJig. Other designs instead provide *traits* [35, 42], which emphasize non-linear composition using rich operations on trait members. OCaml's addition of first-class modules [19] enables a kind of run-time class composition as well. These designs all provide flexible class composition, but typically do not provide the ability to compose classes at run-time.

6 Lessons Learned and Future Work

This paper explains what it takes to turn a theoretical calculus of gradual typing into a full-fledged object-oriented language that respects pre-existing constraints, especially dynamic class composition idioms. The key insights are the theoretical and practical design principles that are applicable across the board. In addition, the paper introduces a novel performance evaluation framework for gradual typing. While the performance results are restricted to the formative part of our design work, they have confirmed a long-held belief among researchers in the community; no other gradual typing project has reported anything comparable. The current implementation of TR owes its shape to negative results from this evaluation.

Our work suggests two kinds of future efforts. First, we need to scale up the formative performance evaluation to a summative one that uses a variety of programs. We also intend to use the framework on a different gradually typed language, e.g., Reticulated Python, that takes a micro-level approach to gradual typing. Doing so will confirm that this approach is useful across the board. Second, the performance framework also suggests that programmers need tailored performance-measuring tools that help them find a path from slow-performing configurations to better ones. For Typed Racket, we intend to investigate the use of profiling techniques [43] that pinpoint the most expensive boundaries so that programmers can eliminate those first.

Acknowledgments. The authors wish to thank Leif Andersen, Ben Greenman, and Vincent St-Amour for their comments on early drafts and for discussions about the research itself. We also thank the anonymous reviewers for their feedback.

The work was partially supported by a DARPA grant at Northeastern and Utah, an NSA grant at Indiana, and several NSF grants at all four sites.

References

1 M. Abadi and L. Cardelli. A Theory of Objects. Springer-Verlag, 1996.

2 E. Allen, J. Bannet, and R. Cartwright. A First-class Approach to Genericity. In *Proc. OOPSLA*, pp. 96–114, 2003.

3 E. Allende, O. Callaú, J. Fabry, É. Tanter, and M. Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, 2013.

4 E. Allende, J. Fabry, and É. Tanter. Cast Insertion Strategies for Gradually-Typed Objects. In *Proc. DLS*, pp. 27–36, 2013.

5 J. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic Inference of Static Types for Ruby. In *Proc. POPL*, pp. 459–472, 2011.

6 D. Ancona, G. Lagorio, and E. Zucca. Jam – A Smooth Extension of Java with Mixins. In *Proc. ESOP*, pp. 154–178, 2000.

7 B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn: Robust, Concurrent, Extensible Scripting on the JVM. In *Proc. OOPSLA*, pp. 117–136, 2009.

8 G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proc. OOPSLA*, pp. 215–230, 1993.

9 R. Cartwright and M. Fagan. Soft Typing. In *Proc. PLDI*, pp. 278–292, 1991.

10 R. Chugh, D. Herman, and R. Jhala. Dependent Types for JavaScript. In *Proc. OOPSLA*, pp. 587–606, 2012.

11 R. L. Constable and N. P. Mendler. Recursive Definitions in Type Theory. Cornell University, TR 85-659, 1985.

12 M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. A Functional I/O System (or Fun for Freshman Kids). In *Proc. ICFP*, pp. 47–58, 2009.
13 C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *Proc. PLDI*, pp. 23–32, 1996.
14 M. Flatt, E. Barzilay, and R. B. Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *Proc. ICFP*, pp. 109–120, 2009.
15 M. Flatt, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Programming Languages as Operating Systems (or Revenge of the Son of the Lisp Machine). In *Proc. ICFP*, pp. 138–147, 1999.
16 M. Flatt, R. B. Findler, and M. Felleisen. Scheme with Classes, Mixins, and Traits. In *Proc. APLAS*, pp. 270–289, 2006.
17 M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *Proc. POPL*, pp. 171–183, 1998.
18 M. Flatt and PLT. Reference: Racket. PLT Inc., PLT-TR-2010-1, 2010. `http://racket-lang.org/tr1/`
19 A. Frisch and J. Garrique. First-class Modules and Composable Signatures in Objective Caml 3.12. In *Proc. ML Workshop*, 2010.
20 M. Furr, J. An, J. S. Foster, and M. Hicks. Static Type Inference for Ruby. In *Proc. SAC*, pp. 1859–1866, 2009.
21 M. Greenberg. Space-Efficient Manifest Contracts. In *Proc. POPL*, pp. 181–194, 2015.
22 S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik. An Empirical Study on the Impact of Static Typing on Software Maintainability. *Empirical Software Engineering*, pp. 1–48, 2013.
23 F. Henglein. Dynamic Typing: Syntax and Proof Theory. *Science of Computer Programming* 22(3), pp. 197–230, 1994.
24 F. Henglein and J. Rehof. Safe Polymorphic Type Inference for a Dynamically Typed Language: Translating Scheme to ML. In *Proc. FPCA*, pp. 192–203, 1995.
25 D. Herman, A. Tomb, and C. Flanagan. Space-efficient Gradual Typing. *HOSC* 23(2), pp. 167–189, 2010.
26 L. Ina and A. Igarashi. Gradual Typing for Generics. In *Proc. OOPSLA*, pp. 609–624, 2011.
27 T. Kamina and T. Tamai. McJava – A Design and Implementation of Java with Mixin-Types. In *Proc. APLAS*, pp. 398–414, 2004.
28 S. Marlow and P. Wadler. A Practical Subtyping System for Erlang. In *Proc. ICFP*, pp. 136–149, 1997.
29 J. Matthews and R. B. Findler. Operational Semantics for Multi-Language Programs. *TOPLAS* 31(3), pp. 12:1–12:44, 2009.
30 S. McDirmid, M. Flatt, and W. C. Hsleh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proc. OOPSLA*, pp. 211–222, 2001.
31 P. Meunier, R. B. Findler, and M. Felleisen. Modular Set-Based Analysis from Contracts. In *Proc. POPL*, pp. 218–231, 2006.
32 A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Proc. POPL*, pp. 167–180, 2015.
33 B. M. Ren, J. Toman, T. S. Strickland, and J. S. Foster. The Ruby Type Checker. In *Proc. SAC*, pp. 1565–1572, 2013.
34 D. Rémy. Programming Objects with ML-ART an Extension to ML with Abstract and Record Types. In *Proc. TACS*, pp. 321–346, 1994.
35 N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable Units of Behaviour. In *Proc. ECOOP*, pp. 248–274, 2003.
36 M. Scriven. The Methodology of Evaluation. Perspectives of Curriculum Evaluation. Rand McNally, 1967.

37 M. Servetto and E. Zucca. MetaFJig: a Meta-circular Composition Language for Java-like Classes. In *Proc. OOPSLA*, pp. 464–483, 2010.

38 J. G. Siek, M. M. Vitousek, M. Cimmini, S. Tobin-Hochstadt, and R. Garcia. Monotonic References for Efficient Gradual Typing. In *Proc. ESOP*, pp. 432–456, 2015.

39 J. G. Siek and W. Taha. Gradual Typing for Functional Languages. In *Proc. SFP*, 2006.

40 J. G. Siek and W. Taha. Gradual Typing for Objects. In *Proc. ECOOP*, pp. 2–27, 2007.

41 J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Proc. POPL*, pp. 365–376, 2010.

42 C. Smith and S. Drossopoulou. Chai: Traits for Java-Like Languages. In *Proc. ECOOP*, pp. 453–478, 2005.

43 V. St-Amour, L. Andersen, and M. Felleisen. Feature-specific profiling. In *Proc. CC*, pp. 49–68, 2015.

44 T. S. Strickland and M. Felleisen. Contracts for First-Class Classes. In *Proc. DLS*, pp. 97–112, 2010.

45 T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In *Proc. OOPSLA*, pp. 943–962, 2012.

46 A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual Typing for First-Class Classes. In *Proc. OOPSLA*, pp. 793–810, 2012.

47 S. Tobin-Hochstadt. Typed Scheme: From Scripts to Programs. Ph.D. dissertation, Northeastern University, 2010.

48 S. Tobin-Hochstadt and M. Felleisen. Interlanguage Migration: from Scripts to Programs. In *Proc. DLS*, pp. 964–974, 2006.

49 S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *Proc. POPL*, pp. 395–406, 2008.

50 S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as Libraries. In *Proc. PLDI*, pp. 132–141, 2011.

51 Typescript Language Specification. Microsoft, Version 0.9.1, 2013.

52 M. M. Vitousek, A. Kent, J. G. Siek, and J. Baker. Design and Evaluation of Gradual Typing for Python. In *Proc. DLS*, pp. 45–56, 2014.

53 R. Wolff, R. Garcia, É. Tanter, and J. Aldritch. Gradual Typestate. In *Proc. ECOOP*, pp. 459–483, 2011.

54 A. K. Wright and R. Cartwright. A Practical Soft Type System for Scheme. *TOPLAS* 19(1), pp. 87–152, 1997.

55 T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating Typed and Untyped Code in a Scripting Language. In *Proc. POPL*, pp. 377–388, 2010.

TreatJS: Higher-Order Contracts for JavaScript

Matthias Keil and Peter Thiemann

Institute for Computer Science
University of Freiburg
Freiburg, Germany
{keilr,thiemann}@informatik.uni-freiburg.de

Abstract

TreatJS is a language embedded, higher-order contract system for JavaScript which enforces contracts by run-time monitoring. Beyond providing the standard abstractions for building higher-order contracts (base, function, and object contracts), *TreatJS*'s novel contributions are its guarantee of non-interfering contract execution, its systematic approach to blame assignment, its support for contracts in the style of union and intersection types, and its notion of a parameterized contract scope, which is the building block for composable run-time generated contracts that generalize dependent function contracts.

TreatJS is implemented as a library so that all aspects of a contract can be specified using the full JavaScript language. The library relies on JavaScript proxies to guarantee full interposition for contracts. It further exploits JavaScript's reflective features to run contracts in a sandbox environment, which guarantees that the execution of contract code does not modify the application state. No source code transformation or change in the JavaScript run-time system is required. The impact of contracts on execution speed is evaluated using the Google Octane benchmark.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Higher-Order Contracts, JavaScript, Proxies

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.28

Supplementary Material ECOOP Artifact Evaluation approved artifact available at
`http://dx.doi.org/10.4230/DARTS.1.1.1`

1 Introduction

A contract specifies the interface of a software component by stating obligations and benefits for the component's users. Customarily contracts comprise invariants for objects and components as well as pre- and postconditions for individual methods. Prima facie such contracts may be specified using straightforward assertions. But further contract constructions are needed for contemporary languages with first-class functions and other advanced abstractions. These facilities require higher-order contracts as well as ways to dynamically construct contracts that depend on run-time values.

Software contracts were introduced with Meyer's *Design by Contract*™ methodology [39] that stipulates the specification of contracts for all components of a program and the monitoring of these contracts while the program is running. Since then, the contract idea has taken off and systems for contract monitoring are available for many languages [33, 1, 37, 32, 12, 22, 11, 10] and with a wealth of features [35, 31, 7, 20, 46, 16, 2]. Contracts are particularly important for dynamically typed languages as these languages only provide memory safety and dynamic type safety. Hence, it does not come as a surprise that the first higher-order contract systems were devised for Scheme and Racket [24], out of the need to

© Matthias Keil and Peter Thiemann;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 28–51

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

create maintainable software. Other dynamic languages like JavaScript[1], Python[2], Ruby[3], PHP[4], and Lua[5] have followed suit.

Many contract systems [33, 12, 22, 10, 35, 46, 16, 2] are *language-embedded*: contracts are first-class values constructed through some library API. This approach is advantageous because it does not tie the contract system to a particular implementation, it neither requires users to learn a separate contract language nor do implementors have to develop specialized contract tools. As the contract system can be distributed as a library, it is easily extensible.

But there are also disadvantages because the contract execution may get entangled with the application code. For example, every contract system supports "flat" contracts which assert that a predicate holds for a value. In most language-embedded systems, the predicate is just a host-language function returning a boolean value. Unlike a real predicate, such a function may have side effects that change the behavior of the application code.

Contributions

We present the design and implementation of *TreatJS*, a language embedded, higher-order contract system for JavaScript [21] which enforces contracts by run-time monitoring. *TreatJS* supports most features of existing systems and a range of novel features that have not been implemented in this combination before. No source code transformation or change in the JavaScript run-time system is required. In particular, *TreatJS* is the first contract system for JavaScript that supports the standard features of contemporary contract systems (embedded contract language, JavaScript in flat contracts, contracts as projections, full interposition using JavaScript proxies [47]) in combination with the following three novel points.

1. *Noninterference.* Contracts are guaranteed not to exert side effects on a contract abiding program execution. A predicate is an arbitrary JavaScript function, which can access the state of the application program but which cannot change it. An exception thrown by a predicate is not visible to the application program. Our guarantees are explained in detail in Section 4.3.
2. *Dynamic contract construction.* Contracts can be constructed and composed at run time using contract abstractions *without compromising noninterference.* A contract abstraction may contain arbitrary JavaScript code; it may read from global state and it may maintain encapsulated local state. The latter feature can be used to construct recursive contracts lazily or to remember values from the prestate of a function for checking the postcondition.
3. *New contract operators.* Beyond the standard contract constructors (flat, function, pairs), *TreatJS* supports object, intersection, and union contracts. Furthermore, contracts can be combined arbitrarily with the boolean connectives: conjunction, disjunction, and negation.

The discussion of related work in Section 6 contains a detailed comparison with other systems. The implementation of the system is available on the Web[6].

---

[1] `http://kinsey.no/projects/jsContract/`, `https://github.com/disnet/contracts.js`
[2] `http://legacy.python.org/dev/peps/pep-0316/`
[3] `https://github.com/egonSchiele/contracts.ruby`
[4] `https://github.com/wick-ed/php-by-contract`
[5] `http://luaforge.net/projects/luacontractor/`
[6] `http://proglang.informatik.uni-freiburg.de/treatjs/`

Overview

The rest of this paper is organized as follows: Section 2 introduces *TreatJS* from a programmer's point of view. Section 3 specifies contract monitoring and Section 4 explains the principles underlying the implementation. Section 5 reports our experiences from applying *TreatJS* to a range of benchmark programs. Section 6 discusses related work and Section 7 concludes.

A technical report [36] extends this paper by an appendix with further technical details, examples, and a formalization of contracts and contract monitoring.

2 A *TreatJS* Primer

The design of *TreatJS* obeys the following rationales.

- *Simplicity and orthogonality.* A core API provides the essential features in isolation. While complex contracts may require using the core API, the majority of contracts can be stated in terms of a convenience API that *TreatJS* provides on top of the core.
- *Non-interference.* Contract checking does not interfere with contract abiding executions of the host program.
- *Composability.* Contracts can be composed arbitrarily.

A series of examples explains how contracts are written and how contract monitoring works. The contract API includes *constructors* that build contracts from other contracts and auxiliary data as well as an ***assert*** function that attaches a contract to a JavaScript value.

Our discussion focuses on general design issues for contracts and avoids JavaScript specifics where possible. Contracts.js [18] provides contracts tailored to the idiosyncrasies of JavaScript's object system – these may be added to *TreatJS* easily.

2.1 Base Contracts

The base contract (aka flat contract) is the fundamental building block for all other contracts. It is defined by a predicate and asserting it to a value immediately sets it in action. We discuss it for completeness – all contract libraries that we know of provide this functionality, but they do not guarantee noninterference like *TreatJS* does.

In JavaScript, any function can be used as a predicate, because any return value can be converted to boolean. JavaScript programmers speak of *truthy* or *falsy* about values that convert to true or false. Thus, a predicate holds for a value if applying the function evaluates to a truthy value.

For example, the function *typeOfNumber* checks its argument to be a number. We apply the appropriate contract constructor to create a base contract from it.

```
1 function typeOfNumber (arg) {
2   return (typeof arg) === 'number';
3 };
4 var typeNumber = Contract.Base(typeOfNumber);
```

Contract is the object that encapsulates the *TreatJS* implementation. Its ***assert*** function attaches a contract to a value. Attaching a base contract causes the predicate to be checked immediately. If the predicate holds, ***assert*** returns the original value. Otherwise, ***assert*** signals a contract violation blaming the *subject*.

In the following example, the first ***assert*** returns *1* whereas the second ***assert*** fails.

```
7  var typeBoolean = Contract.Base(function (arg) {
8    return (typeof arg) === 'boolean';
9  });
10 var typeString = Contract.Base(function (arg) {
11   return (typeof arg) === 'string';
12 });
13 var isArray = Contract.With({Array:Array},
14   Contract.Base(function (arg) {
15     return (arg instanceof Array);
16 }));
```

Listing 1 Some utility contracts.

```
5 Contract.assert(1, typeNumber); // accepted
6 Contract.assert('a', typeNumber); // violation, blame subject 'a'
```

Listing 1 defines a number of base contracts for later use. Analogous to *typeNumber*, the contracts *typeBoolean* and *typeString* check the type of their argument. Contract *isArray* checks if the argument is an array. Its correct implementation requires the ***With*** operator, which will be explained in Section 2.4.

2.2 Higher-Order Contracts

The example contracts in Subsection 2.1 are geared towards values of primitive type, but a base contract may also specify properties of functions and other objects. However, base contracts are not sufficiently expressive to state interesting properties of objects and functions. For example, a contract should be able to express that a function maps a number to a boolean or that a certain field access on an object always returns a number.

2.2.1 Function Contracts

Following Findler and Felleisen [24], a function contract is built from zero or more contracts for the arguments and one contract for the result of the function. Asserting the function contract amounts to asserting the argument contracts to the arguments of each call of the function and to asserting the result contract to the return value of each call. Asserting a function contract to a value immediately signals a contract violation if the value is not a function. Nevertheless, we call a function contract *delayed*, because asserting it to a function does not immediately signal a contract violation.

As a running example, we develop several contracts for the function *cmpUnchecked*, which compares two values and returns a boolean.

```
17 function cmpUnchecked(x, y) {
18   return (x > y);
19 }
```

Our first contract restricts the arguments to numbers and asserts that the result of the comparison is a boolean.

```
20 var cmp = Contract.assert(cmpUnchecked,
21                  Contract.AFunction([typeNumber,typeNumber],typeBoolean));
```

AFunction is the convenience constructor for a function contract. Its first argument is an array with n contracts for the first n function arguments and the last argument is the contract for the result. This contract constructor is sufficient for most functions that take a fixed number of arguments.

The contracted function accepts arguments that satisfies contract *typeNumber* and promises to return a value that satisfies *typeBoolean*. If there is call with an argument that violates its contract, then the function contract raises an exception blaming the *context*, which is the caller of the function that provides the wrong kind of argument. If the argument is ok but the result fails, then blame is assigned to the *subject* (i.e., the function itself). Here are some examples that exercise *cmp*.

```
22 cmp(1,2); // accepted
23 cmp('a','b'); // violation, blame the context
```

To obtain a subject violation we use a broken version of *cmpUnchecked* that sometimes returns a string.

```
24 var cmpBroken = function(x, y) {
25   return (x>0 && y>0) ? (x > y) : 'error';
26 }
27 var faultyCmp = Contract.assert(cmpBroken,
28                        Contract.AFunction([typeNumber,typeNumber],typeBoolean));
29 faultyCmp(0,1); // violation, blame the subject
```

Higher-order contracts may be defined in the usual way and their blame reporting in *TreatJS* follows Findler and Felleisen [24]. For example, a function *sort*, which takes an array and a numeric comparison function as arguments and which returns an array, may be specified by the following contract, which demonstrates nesting of function contracts.

```
30 var sortNumbers = Contract.AFunction([isArray, cmp], isArray);
```

Higher-order contracts open up new ways for a function not to fulfill its contract. For example, *sort* may violate the contract by calling its comparison function (contracted with *cmp*) with non-numeric arguments. Generally, the context is responsible to pass an argument that satisfies its specification to the function and to use the function's result according to its specification. Likewise, the function is responsible for the use of its arguments and in case the arguments meet their specification to return a value that conforms to its specification.

In general, a JavaScript function has no fixed arity and arguments are passed to the function in a special array-like object, the *arguments* object. Thus, the core contract ***Function*** takes two arguments. The first argument is an object contract (cf. Subsubsection 2.2.2) that maps an argument index (starting from zero) to a contract. The second argument is the contract for the function's return value. Thus, ***AFunction*** creates an object contract from the array in its first argument and passes it to ***Function***.

Using the core ***Function*** contract is a bit tricky because it exposes the unwary contract writer to some JavaScript internals. The contract ***Function****(isArray, typeNumber)* checks whether the arguments object is an array (which it is not), but it does *not* check the function's arguments. As a useful application of this feature, the following contract *twoArgs* checks that a function is called with exactly two arguments.

```
31 var lengthTwo = Contract.Base(function (args) {
32   return (args.length == 2);
33 });
34 var Any = Contract.Base (function() { return true; });
35 var twoArgs = Contract.Function(lengthTwo, any);
```

2.2.2 Object Contracts

Apart from base contracts that are checked immediately and delayed contracts for functions, *TreatJS* provides contracts for objects. An object contract is defined by a mapping from property names to contracts. Asserting an object contract to a value immediately signals a violation if the value is not an object. The contracts in the mapping have no immediate effect. However, when reading a property of the contracted object, the contract associated with this property is asserted to the property value. Similarly, when writing a property, the new value is checked against the contract. This way, each value read from a property and each value that is newly written into the property is guaranteed to satisfy the property's contract. Reads and writes to properties not listed in an object contract are not checked.

The following object contract indicates that the *length* property of an object is a number. The constructor ***AObject*** expects the mapping from property names to contracts as a JavaScript object.

```
36 var arraySpec = Contract.AObject({length:typeNumber});
```

Any array object would satisfy this contract. Each access to the *length* property of the contracted array would be checked to satisfy *typeNumber*.

Blame assignment for property reads and writes is inspired by Reynolds [43] interface for a reference cell: each property is represented as a pair of a getter and a setter function. Both, getter and setter apply the same contract, but they generate different blame. If the contract fails in the getter, then the *subject* (i.e., the object) is blamed. If the contract fails in the setter, then the *context* (i.e., the assignment) is blamed. The following example illustrates this behavior.

```
37 var faultyObj = Contract.assert({length:'1'}, arraySpec);
38 faultyObj.length; // violation, blame the subject
39 faultyObj.length='1'; // violation, blame the context
```

An object contract may also serve as the domain portion in a function contract. It gives rise to yet another equivalent way of writing the contract from Line 21.

```
40 Contract.Function(
41   Contract.AObject([typeNumber, typeNumber]), typeBoolean);
```

Functions may also take an intersection (cf. Section 2.3) of a function contract and an object contract to address properties of functions and *this*. There is also a special ***Method*** contract that includes a contract specification for *this*.

2.3 Combination of Contracts

Beyond base, function, and object contracts, *TreatJS* provides the intersection and union of contracts as well as the standard boolean operators on contracts: conjunction (***And***), disjunction (***Or***), and negation (***Not***). The result of an operator on contracts is again a contract that may be further composed.

For space reasons, we only discuss intersection and union contracts, which are inspired by the corresponding operators in type theory. If a value has two types, then we can assign it an *intersection type* [13]. It is well known that intersection types are useful to model overloading and multiple inheritance.

As an example, we revisit *cmpUnchecked*, which we contracted with *cmpNumbers* in Section 2.2.1 to ensure that its arguments are numbers. As the comparison operators are overloaded to work for strings, too, the following contract is appropriate.

```
42 Contract.Intersection(
43   Contract.AFunction([typeNumber, typeNumber], typeBoolean),
44   Contract.AFunction([typeString, typeString], typeBoolean));
```

This contract blames the context if the contracted function is applied to arguments that fail both domain contracts, that is, *[typeNumber, typeNumber]* and *[typeString, typeString]*. The subject is blamed if a function call does not fulfill the range contract that corresponds to a satisfied domain contract.

This interpretation coincides nicely with the meaning of an intersection type. The caller may apply the function to arguments both satisfying either *typeNumber* or *typeString*. In general, the argument has to satisfy the union of *typeNumber* and *typeString*. For disjoint arguments the intersection contract behaves identically to the disjunction contract.

As in type theory, the union contract is the dual of an intersection contract. Exploiting the well-known type equivalence $(A \to C) \wedge (B \to C) = (A \vee B) \to C$ [5], we may rephrase the above contract with a union contract, which accepts either a pair of numbers or a pair of strings as function arguments:

```
45 Contract.Function(
46   Contract.Union(
47     Contract.AObject([typeNumber, typeNumber]),
48     Contract.AObject([typeString, typeString])),
49  typeBoolean);
```

Next, we consider the union of two function contracts.

```
50 var uf = Contract.Union(
51   Contract.AFunction([typeNumber,typeNumber], typeBoolean),
52   Contract.AFunction([typeString,typeString], typeBoolean));
```

Asserting this contract severely restricts the domain of a function. An argument is only acceptable if it is acceptable for all function contracts in the union. Thus, the context is blamed if it provides an argument that does not fulfill both constituent contracts. For example, *uf* requires an argument that is both a number and a string. As there is no such argument, any caller will be blamed.

For a sensible application of a union of function contracts, the domains should overlap:

```
53 Contract.Union(
54   Contract.AFunction([typeNumber, typeNumber], typeBoolean),
55   Contract.AFunction([typeNumber, typeNumber], typeString));
```

This contract is satisfied by a function that either always returns a boolean value or by one that always returns an string value. It is *not* satisfied by a function that alternates between both return types between calls. A misbehaving function is blamed on the first alternation.

2.4 Sandboxing Contracts

All contracts of *TreatJS* guarantee noninterference: Program execution is not influenced by the evaluation of a terminating predicate inside a base contract. That is, a program with contracts is guaranteed to read the same values and write to the same objects as without contracts. Furthermore, it either signals a contract violation or returns a results that behaves the same as without contracts.

To achieve this behavior, predicates must not write to data structures visible outside of the predicate. For this reason, predicate evaluation takes place in a sandbox that hides all external bindings and places a write protection on objects passed as parameters.

To illustrate, we recap the *typeNumber* contract from Line 4. Without the sandbox we could abstract the target type of *typeNumber* with a function and build base contracts by applying the function to different type names as in the following attempt:

```
56 function badTypeOf(type) {
57   return Contract.Base(function(arg) {
58     return (typeof arg) === type;
59   });
60 }
61 var typeNumberBad=badTypeOf('number');
62 var typeStringBad=badTypeOf('string');
```

However, this code fragment does not work as expected. The implementation method for our sandbox reopens the closure of the anonymous function in line 57 and removes the binding for *type* from the contract's predicate. Both *typeNumberBad* and *typeStringBad* would be stopped by the sandbox because they try to access the (apparently) global variable *type*. This step is required to guarantee noninterference, because the syntax of predicates is not restricted in their expressiveness and programmers may do arbitrary things, including communicating via global variables or modifying data outside the predicate's scope.

In general, read-only access to data (functions and objects) is safe and many useful contracts (e.g., the *isArray* contract from Line 14 references the global variable *Array*) require access to global variables, so a sandbox should permit regulated access.

Without giving specific permission, the sandbox rejects *any* access to the *Array* object and signals a sandbox violation. To grant read permission, a new contract operator ***With*** is needed that makes an external reference available inside the sandbox. The ***With*** operator takes a *binding object* that maps identifiers to values and a contract. Evaluating the resulting contract installs the binding in the sandbox environment and then evaluates the constituent contract with this binding in place. Each value passed into the sandbox (as an argument or as a binding) is wrapped in an identity preserving membrane [47] to ensure read-only access to the entire object structure.

The ***With*** constructor is one approach to build parameterized contracts by providing a form of dynamic binding.

```
63 var typeOf = Contract.Base(function(arg)  {
64   return (typeof arg) === type;
65 });
66 var typeNumber=Contract.With({type:'number'},typeOf);
67 var typeString=Contract.With({type:'string'},typeOf);
```

For aficionados of lexical scope, contract constructors, explained in the next subsection, are another means for implementing parameterized contracts.

2.5 Contract Constructors

While sandboxing guarantees noninterference, it prohibits the formation of some useful contracts. For example, the range portion of a function contract may depend on the arguments or a contract may enforce a temporal property by remembering previous function

calls or previously accessed properties. Implementing such a facility requires that predicates should be able to store data without affecting normal program execution.

TreatJS provides a *contract constructor* ***Constructor*** for building a parameterized contract. The constructor takes a function that maps the parameters to a contract. This function is evaluated in a sandbox, like a predicate. Unlike a predicate, the function may contain contract definitions and must return a contract. Each contract defined inside the sandbox is associated with the same sandbox environment and shares the local variables and the parameters visible in the function's scope. No further sandboxing is needed for the predicates / base contracts defined inside the sandbox. The returned contract has no ties to the outside world and thus the included predicates will not be evaluated in the sandbox again. If such a predicate is called, the encapsulated sandbox environment can be used to store data for later use and without affecting normal program execution.

In the next example, a contract constructor builds a base contract from the name of a type. The constructor provides a lexically scoped alternative to the approach in Line 63.

```
68 var Type = Contract.Constructor(function(type) {
69   return Contract.Base(function(arg) {
70     return (typeof arg) === type;
71   });
72 });
```

To obtain the actual contract we apply the constructor to parameters with the method ***Contract.construct****(Type, 'number')* or by using the ***construct*** method of the constructor.

```
73 var typeNumber = Type.construct('number');
74 var typeString = Type.construct('string');
```

Let's consider yet another contract for a compare function. For this contract, we only want the contract of the comparison to state that the two arguments have the same type.

```
75 Contract.Constructor(function() {
76   var type;
77   var getType = Contract.Base(function (arg) {
78     return type = (typeof arg);
79   });
80   var checkType = Contract.Base(function (arg) {
81     return type === (typeof arg);
82   });
83   var typeBoolean = Contract.Base(function (arg) {
84     return (typeof arg) === 'boolean';
85   });
86   return Contract.AFunction([getType, checkType], typeBoolean);
87 });
```

This code fragment defines a constructor with zero parameters (viz. the empty parameter list in Line 75). As there are no parameters, this example only uses the constructor to install a shared scope for several contracts. The contract *getType* saves the type of the first argument. The comparison function has to satisfy a function contract which compares the type of the second arguments with the saved type.

2.6 Dependent Contracts

A dependent contract is a contract on functions where the range portion depends on the function argument. The contract for the function's range can be created with a contract constructor. This constructor is invoked with the caller's argument. Additionally, it is possible to import pre-state values in the scope of the constructor so that the returned contract may refer to those values.

TreatJS's dependent contract operation only builds a range contract in this way; it does not check the domain as checking the domain may be achieved by conjunction with another function contract. By either pre- or postcomposing the other contract, the programmer may choose between picky and lax semantics for dependent contracts (cf. [29]).

For example, a dependent contract *PreserveLength* may specify that an array processing function like *sort* (Line 30) preserves the length of its input. The constructor receives the arguments (input array and comparison function) of a function call and returns a contract for the range that checks that the length of the input array is equal to the length of the result.

```
88 var PreserveLength = Contract.Dependent(
89   Contract.Constructor(function(input, cmp) {
90     return Contract.Base(function (result) {
91       return (input.length === result.length);
92     });
93 }));
```

3 Contract Monitoring

This section explains how contract monitoring works and how the outcome of a contract assertion is determined by the outcome of its constituents. For space reasons we focus on the standard contract types (base, function, and object contracts) with intersection and union; we describe the boolean operators in the supplemental material.

3.1 Contracts and Normalization

A contract is either an immediate contract, a delayed contract, an intersection between an immediate and a delayed contract, or a union of contracts. Immediate contracts may be checked right away when asserted to a value whereas delayed contracts need to be checked later on. Only a base contract is immediate.

A delayed contract is a function contract, a dependent contract, an object contract, or an intersection of delayed contracts. Intersections are included because all parts of an intersection must be checked on each use of the contracted object: a call to a function or an access to an object property.

The presence of operators like intersection and union has severe implications. In particular, a failing base contract must not signal a violation immediately because it may be enclosed in an intersection. Reporting the violation must be deferred until the enclosing operator is sure to fail.

To achieve the correct behavior for reporting violations, monitoring *normalizes* contracts before it starts contract enforcement. Normalization separates the immediate parts of a contract from its delayed parts so that each immediate contract can be evaluated directly,

whereas the remaining delayed contracts wrap the subject of the contract in a proxy that asserts the contract when the subject is used.

To expose the immediate contracts, normalization first pulls unions out of intersections by applying the distributive law suitably. The result is a union of intersections where the operands of each intersection are either immediate contracts or function contracts. At this point, monitoring can check all immediate contracts and set up proxies for the remaining delayed contracts. It remains to define the structure needed to implement reporting of violations (i.e., blame) that is able to deal with arbitrary combinations of contracts.

3.2 Callbacks

To assert a contract correctly, its evaluation must connect each contract with the enclosing operations and it must keep track of the evaluation state of these operations. In general, the signaling of a violation depends on a combination of failures in different contracts.

This connection is modeled by so-called *callbacks*. They are tied to a particular contract assertion and link each contract to its next enclosing operation or, at the top-level, its assertion. A callback linked to a source-level assertion is called *root callback*. Each callback implements a constraint that specifies the outcome of a contract assertion in terms of its constituents.

A callback is implemented as a method that accepts the result of a contract assertion. The method updates a shared property, it evaluates the constraint, and passes the result to the enclosing callback.

Each callback is related to one specific contract occurrence in the program; there is at least one callback for each contract occurrence and there may be multiple callbacks for a delayed contract (e.g., a function contract). The callback is associated with a record that defines the blame assignment for the contract occurrence. This record contains two fields, *subject* and *context*. The intuitive meaning of the fields is as follows. If the *subject* field is false, then the contract fails blaming the subject (i.e., the value to which the contract is asserted). If the *context* field is false, then the contract fails blaming the context (i.e., the user of the value to which the contract is asserted).

3.3 Blame Calculation

The fields in the record range over $\mathbb{B}_4$, the lattice underlying Belnap's four-valued logic [6], which is intended to deal with incomplete or inconsistent information. The set $\mathbb{B}_4 = \{\bot, \mathsf{f}, \mathsf{t}, \top\}$ of truth values forms a lattice modeling accumulated knowledge about the truth of a proposition. Thus, a truth value may be considered as the set of classical truth values $\{\mathit{true}, \mathit{false}\}$ that have been observed so far for a proposition. For instance, contracts are valued as $\bot$ before they are evaluated and $\top$ signals potentially conflicting outcomes of repeated checking of the same contract.

As soon as a base contract's predicate returns, the contract's callback is applied to its outcome. A function translates the outcome to a truth value according to JavaScript's idea of *truthy* and *falsy*, where *false*, *undefined*, *null*, *NaN*, and *""* is interpreted as false. Exceptions thrown during evaluation of a base contract are captured and count as $\top$.

A new contract assertion signals a violation if a root callback maps any field to f or $\top$. Evaluation continues if only internal fields have been set to f or $\top$.

3.4 Contract Assertion

Contract monitoring starts when calling the assert function with a value and a contract.

The top-level assertion first creates a new root callback that may signal a contract violation later on and an empty sandbox object that serves as the context for the internal contract monitoring. The sandbox object carries all external values visible to the contract.

Asserting a base contract to a value wraps the value to avoid interference and applies the predicate to the wrapped value. Finally, it applies the current callback function to the predicate's outcome.

Asserting a delayed contract to an object results in a proxy object that contains the current sandbox object, the associated callback, and the contract itself. It is an error to assert a delayed contract to a primitive value.

Asserting a union contract first creates a new callback that combines the outcomes of the subcontracts according to the blame propagation rules for the union. Then it asserts the first subcontract (with a reference to one input of the new callback) to the value before it asserts the second subcontract (with a reference to the other input) to the resulting value.

Asserting a ***With*** contract first wraps the values defined in the binding object and builds a new sandbox object by merging the resulting values and the current sandbox object. Then it asserts its subcontract.

3.5 Application, Read, and Assignment

Function application, property read, and property assignment distinguish two cases: either the operation applies directly to a non-proxy object or it applies to a proxy. If the target of the operation is not a proxy object, then the standard operation is applied.

If the target object is a proxy with a delayed contract, then the contract is checked when the object is used as follows.

A function application on a contracted function first creates a fresh callback that combines the outcomes of the argument and range contract according to the propagation rules for functions. Then it asserts the domain contract to the argument object with reference to the domain input of the new callback before it applies the function to the result. After completion, the range contract is applied to the function's result with reference to the range input of the callback.

A function application on a function contracted with a dependent contract first applies the contract constructor to the argument and saves the resulting range contract. Next, it applies the function to the argument and asserts the computed range contract to the result.

A property access on a contracted object has two cases depending on the presence of a contract for the accessed property. If a contract exists, then the contract is asserted to the value after reading it from the target object and before writing it to the target object. Otherwise, the operation applies directly to the target object.

Property write creates a fresh callback that inverts the responsibility of the contract assertion (the context has to assign a value according to the contract).

An operation on a proxy with an intersection contract asserts the first subcontract to the value before it asserts the second subcontract to the resulting value. Both assertions are connected to one input channel of a new callback that combines there outcomes according to the rules for intersection.

All contract assertions forward the sandbox object in the proxy to the subsequent contract assertion.

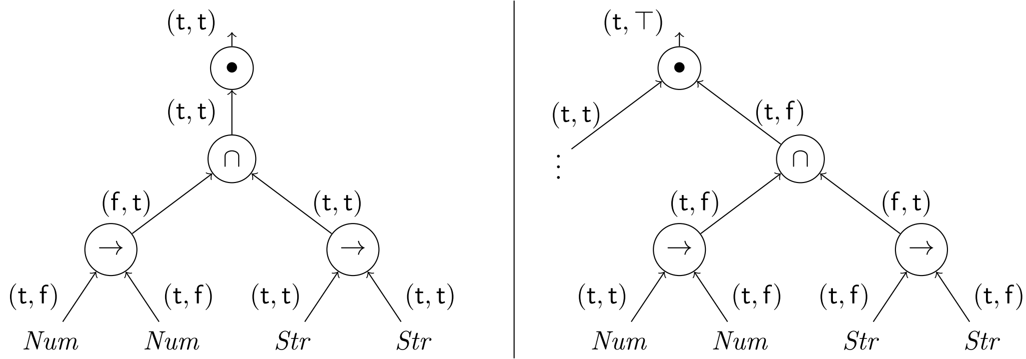


Figure 1 Blame Calculation of $addOne = \lambda x.(x{+}'1')$ contracted with $(Num \to Num) \cap (Str \to Str)$. The left side shows the callback graph after applying *addOne* to the string $'1'$ (first call). The right side shows the graph constructed after applying *addOne* to the number *1* (second call). Each node is a callback. Each edge an input channel. The labeling next to the arrow shows the record (*context*, *subject*). The root callback (●) collects the outcome of all delayed contract assertions.

3.6 Sandbox Encapsulation

The sandbox ensures noninterference with the actual program code by stopping evaluation of the predicate if it attempts to modify data that is visible outside the contract.[7]

To ensure noninterference, the subject of a base contract, the argument of a dependent contract as well as the value passed by a contract constructor are all wrapped in a membrane to ensure that the contract's code cannot modify them in any way.

To wrap a non-proxy object, the object is packaged in a fresh proxy along with the current sandbox object. This packaging ensures that further access to the wrapped object uses the current sandbox object. If the object points to a contracted object, the wrap operation continues with the target object, before adding all contracts from the contracted object. A primitive value is not wrapped.

A property read on a sandboxed object forwards the operation to the target and wraps the resulting behavior. An assignment to a sandboxed object is not allowed, thus it signals a sandbox violation.

The application of a sandboxed function recompiles the function by adding the given sandbox object to the head of the scope chain. Finally, it calls the function. The function's argument and its result are known to be wrapped in this case.

3.7 Contract Satisfaction

The blame assignment for a function contract is calculated from the blame assignment for the argument and result contracts, which are available through the records ι_d and ι_r. A function does not fulfill a contract if it does not fulfill its obligations towards its argument $\iota_d.context$ **or** if the argument fulfills its contract, but the result does not fulfills its contract $\iota_d.subject \Rightarrow \iota_r.subject$. The first part arises for higher-order functions, which may pass illegal arguments to their function-arguments. The second part corresponds to partial correctness of the function with respect to its contract.

A function's context (caller) fulfills the contract if it passes an argument that fulfills its

[7] The sandbox cannot ensure termination of the predicate, of course.

contract $\iota_d.subject$ **and** it uses the function result according to its contract $\iota_r.context$. The second part becomes non-trivial with functions that return functions.

An object (subject) does not fulfill an object contract if a property access returns a value that does not fulfill the contract. An object's context (caller) does not fulfills the contract if it assigns an illegal value to a contracted property **or** it does not uses the objects return according to its contract.

The outcome of read access on a contracted property $\iota_c.subject$ is directly related to the parent callback and does not need a special constraint. A write to a property guarded with contract C generates blame like a call to a function with contract $C \rightarrow Any$. (*Any* accepts any value.)

The blame assignment for an intersection contract is defined from its constituents at ι_r and ι_l. A subject fulfills an intersection contract if it fulfills both constituent contracts: $\iota_r.subject \wedge \iota_l.subject$. A context, however, only needs to fulfill one of the constituent contracts: $\iota_r.context \vee \iota_l.context$.

Dually to the intersection rule, the blame assignment for a union contract is determined from its constituents at ι_l and ι_r. A subject fulfills a union contract if it fulfills one of the constituent contracts: $\iota_l.subject \vee \iota_r.subject$. A context, however, needs to fulfill both constituent contracts: $\iota_l.context \wedge \iota_r.context$, because it does not known which contract is fulfilled by the subject.

Figure 1 illustrates the working of callbacks. After applying *addOne* to *'1'*, the first function contract ($Num \rightarrow Num$) would fail blaming the context, whereas the second contract ($Str \rightarrow Str$) succeeds. Because the context of an intersection may choose which side to fulfill, the intersection is satisfied.

However, the second call which applies *addOne* to *1* raises an exception. The first function contract fails, blaming the subject, whereas the second contract fails, blaming the context. Because the subject of an intersection has to fulfill both contracts, the intersection fails, blaming the subject.

4 Implementation

The implementation is based on the JavaScript Proxy API [47, 48], a part of the ECMAScript 6 draft standard. This API is implemented in Firefox since version 18.0 and in Chrome V8 since version 3.5. Our development is based on the SpiderMonkey JavaScript engine.

4.1 Delayed Contracts

Delayed contracts are implemented using JavaScript Proxies [47, 48], which guarantees full interposition by intercepting all operations. The assertion of a delayed contract wraps the subject of the contract in a proxy. The handler for the proxy contains the contract and implements traps to mediate the use of the subject and to assert the contract. No source code transformation or change in the JavaScript run-time system is required.

4.2 Sandboxing

Our technique to implement sandboxing relies on all the evil and bad parts of JavaScript: the *eval* function and the ***with*** statement. The basic idea is as follows. The standard implementation of the *toString* method of a user-defined JavaScript function returns a string that contains the source code of that function. When *TreatJS* puts a function (e.g., a predicate) in a sandbox, it first *decompiles* it by calling its *toString* method. Applying *eval*

to the resulting string creates a fresh variant of that function, **but** it dynamically rebinds the free variables of the function to whatever is currently in the scope at the call site of *eval*.

JavaScript's ***with*** *(obj){ ... body ... }* statement modifies the current environment by placing *obj* on top of the scope chain while executing *body*. With this construction, which is somewhat related to dynamic binding [30], any property defined in *obj* shadows the corresponding binding deeper down in the scope chain. Thus, we can add and shadow bindings, but we cannot remove them. Or can we?

It turns out that we can also abuse ***with*** to *remove* bindings! The trick is to wrap the new bindings in a proxy object, use ***with*** to put it on top of the scope chain, and to trap the binding object's *hasOwnProperty* method. When JavaScript traverses the scope chain to resolve a variable reference *x*, it calls *hasOwnProperty(x)* on the objects of the scope chain starting from the top. Inside the ***with*** statement, this traversal first checks the proxied binding object. If its *hasOwnProperty* method always returns true, then the traversal stops here and the JavaScript engine sends all read and write operations for free variables to the proxied binding object. This way, we obtain full interposition and the handler of the proxied binding object has complete control over the free variables in *body*.

The ***With*** contract is *TreatJS*'s interface to populate this binding object. The operators for contract abstraction and dependent contracts all take care to stitch the code fragments together in the correct scope. To avoid the frequent decompilation and *eval* of the same code, our implementation caches the compiled code where applicable.

No value is passed inside the sandbox without proper protection. Our protection mechanism is inspired by *Revocable Membranes* [47, 44]. A membrane serves as a regulated communication channel between two worlds, in this case between an object/ a function and the rest of a program. A membrane is essentially a proxy that guards all read operations and – in our case – stops all writes. If the result of a read operation is an object, then it is recursively wrapped in a membrane before it is returned. Access to a property that is bound to a *getter* function needs to decompile the *getter* before its execution. Care is taken to preserve object identities when creating new wrappers (our membrane is *identity preserving*).

We employ membranes to keep the sandbox apart from normal program execution thus guaranteeing noninterference. In particular, we encapsulate objects passed through the membrane, we enforce write protection, and we withhold external bindings from a function.

4.3 Noninterference

The ideal contract system should not interfere with the execution of application code. That is, as long as the application code does not violate any contract, the application should run as if no contracts were present. Borrowing terminology from security, this property is called noninterference (NI) [27]: with the assumption that contract code runs at a higher level of security than application code, the low security application code should not be able to observe the results of the high-level contract computation.

Looking closer, we need to distinguish internal and external sources of interference. Internal sources of interference arise from executing unrestricted JavaScript code in the predicate of a base contract. This code may try to write to an object that is visible to the application, it may throw an exception, or it may not terminate. Our implementation restricts all write operations to local objects using sandboxing. It captures all exceptions and turns them into an appropriate contract outcome. A timeout could be used to transform a contract that may not terminate into an exception, alas, such a timeout cannot be implemented in

JavaScript.[8]

External interference arises from the interaction of the contract system with the language. Two such issues arise in a JavaScript contract system, exceptions and object equality.

Exceptions arise when a contract failure is encoded by a contract exception, as it is done in Eiffel, Racket, and contracts.js. If an application program catches exceptions, then it may become aware of the presence of the contract system by observing an exception caused by a contract violation. Our implementation avoids this problem by reporting the violation and then using a JavaScript API method to quit JavaScript execution[9].

Object equality becomes an issue because function contracts as well as object contracts are implemented by some kind of wrapper. The problem arises if a wrapper is different (i.e., not pointer-equal) from the wrapped object so that an equality test between wrapper and wrapped object or between different wrappers for the same object (read: tests between object and contracted object or between object with contract A and object with contract B) in the application program returns false instead of true.

Our implementation uses JavaScript proxies to implement wrappers. Unfortunately, JavaScript proxies are always different from their wrapped objects and the only safe way to change that is by modifying the proxy implementation in the JavaScript VM. See our companion paper [34] for more discussion. There are proposals based on preprocessing all uses of equality to proxy-dereferencing equality, for example using SweetJS [19], but they do not work in combination with `eval` and hence do not provide full interposition.

5 Evaluation

This section reports on our experience with applying contracts to select programs. We focus on the influence of contract assertion and sandboxing on the execution time.

All benchmarks were run on a machine with two AMD Opteron Processor with 2.20 GHz and 64 GB memory. All example runs and timings reported in this paper were obtained with the SpiderMonkey JavaScript engine.

5.1 Benchmark Programs

To evaluate our implementation, we applied it to JavaScript benchmark programs from the Google Octane 2.0 Benchmark Suite[10]. Octane 2.0 consists of 17 programs that range from performance tests to real-world web applications (Figure 2), from an OS kernel simulation to a portable PDF viewer. Each program focuses on a special purpose, for example, function and method calls, arithmetic and bit operations, array manipulation, JavaScript parsing and compilation, etc.

Octane reports its result in terms of a score. The Octane FAQ[11] explains the score as follows: "*In a nutshell: bigger is better. Octane measures the time a test takes to complete and then assigns a score that is inversely proportional to the run time.*" The constants in this computation are chosen so that the current overall score (i.e., the geometric mean of the individual scores) matches the overall score from earlier releases of Octane and new

[8] The JavaScript *timeout* function only schedules a function to run when the currently running JavaScript code – presumably some event handler – stops. It cannot interrupt a running function.

[9] This aspect is customizable because the API method is not generally available. It can easily be overwritten to report a violation elsewhere or to throw an exception.

[10] `https://developers.google.com/octane/`

[11] `https://developers.google.com/octane/faq`

benchmarks are integrated by choosing the constants so that the geometric mean remains the same. The rationale is to maintain comparability.

5.2 Methodology

To evaluate our implementation, we wrote a source-to-source compiler that first modifies the benchmark code by wrapping each function expression[12] in an additional function. In a first run, this additional function wraps its target function in a proxy that, for each call to the function, records the data types of the arguments and of the function's return value. This recording distinguishes the basic JavaScript data types *boolean*, *null*, *undefined*, *number*, *string*, *function*, and *object*. Afterwards, the wrapper function is used to assert an appropriate function contract to each function expression. These function contracts are built from the types recorded during the first phase. If more than one type is recorded at a given program point, then the disjunction of the individual type contracts is generated.

All run-time measurements were taken from a deterministic run, which requires a predefined number of iterations, and by using a warm-up run.

5.3 Results

Figure 2 contains the scores of all benchmark programs in different configurations, which are explained in the figure's caption. As expected, all scores decrease when adding contracts. The impact of a contract depends on the frequency of its application. A contract on a heavily used function (e.g., in *Richards*, *DeltaBlue*, or *Splay*) causes a significantly higher decrease of the score. These examples show that the run-time impact of contract assertion depends on the program and on the particular value that is monitored. While some programs like *Richards*, *DeltaBlue*, *RayTrace*, and *Splay* are heavily affected, others are almost unaffected: *Crypto*, *NavierStokes*, and *Mandreel*, for instance.

In several cases the execution with contracts (or with a particular feature) is faster than without. All such fluctuations in the score values are smaller than the standard deviation over several runs of the particular benchmark.

For better understanding, Figure 3 lists some numbers of internal counters. The numbers indicate that the heavily affected benchmarks (*Richards*, *DeltaBlue*, *RayTrace*, *Splay*) contain a very large number of internal contract assertions. Other benchmarks are either not affected (*RegExp*, *zlib*) or only slightly affected (*Crypto*, *pdf.js*, *Mandreel*) by contracts.

For example, the *Richards* benchmark performs 24 top-level contract assertions (these are all calls to ***Contract.assert***), 1.6 billion internal contract assertions (including top-level assertions, *delayed* contract checking, and predicate evaluation), and 936 million predicate executions. The sandbox wraps about 4.7 billion elements, but performs only 4 decompile operations. Finally, contract checking performs 3.4 billion callback update operations.

Because of the fluctuation in slightly affected benchmark programs the following discussion focuses on benchmarks that were heavily impacted. Thus, we ignore the benchmark programs *Crypto*, *RegExp*, *pdf.js*, *Mandreel*, *zlib*.

In a first experiment, we turn off predicate execution and return *true* instead of the predicate's result. This splits the performance impact into the impact caused by the contract system (proxies, callbacks, and sandboxing) and the impact caused by evaluating predicates. From the score values we find that the execution of the programmer provided predicates

[12] Function expressions are all expressions of the form ***function**(..){..}*.

Benchmark	F	S	w/o C	w/o D	w/o M	w/o P	B
Richards	0.391	0.519	0.582	0.782	0.781	0.903	11142
DeltaBlue	0.276	0.360	0.409	0.544	0.544	0.625	17462
Crypto	11888	12010	11912	11914	11986	11979	11879
RayTrace	1.09	1.45	1.82	2.51	2.51	3.02	23896
EarleyBoyer	5135	5292	5126	5205	5233	5242	5370
RegExp	1208	1181	1205	1199	1212	1178	1207
Splay	20.6	27.8	31.2	42.5	42.5	49.7	9555
SplayLatency	73.1	99.7	109	151	151	177	6289
NavierStokes	6234	7159	7924	9176	8943	9456	12612
pdf.js	9191	9257	9548	9156	9222	9152	9236
Mandreel	12555	12542	12586	12549	12346	12431	12580
MandreelLatency	18741	18883	18741	18883	19027	18955	19398
Gameboy Emulator	6.80	9.07	10.8	14.9	14.9	17.7	23801
Code loading	6245	6785	6937	7372	7335	7533	9324
Box2DWeb	3.57	4.67	5.72	7.80	7.82	9.19	12528
zlib	29108	28708	29025	29047	28926	29063	29185
TypeScript	187	248	290	400	396	463	11958

Figure 2 Scores for the Google Octane 2.0 Benchmark Suite (bigger is better). Column **F** (*Full*) contains the scores for running with sandboxed contract assertion. Column **S** (*System only*) contains the score values for running TreatJS without predicate evaluation (all predicates are set to *true*) but with all internal components (callback, decompile, membrane). Column **w/o C** (*without callback*) shows the scores from a full run (with predicates) but without callback updates. Column **w/o D** (*without decompile*) shows the scores without recompiling functions. Column **w/o M** (*without membrane*) lists the scores with contract assertion but without sandboxing (and thus without decompile). Column **w/o P** (*without predicate*) shows the score values of raw contract assertions without predicate evaluation and thus without sandboxing, decompile, and callback updates. The last column **B** (*Baseline*) gives the baseline scores without contract assertion.

causes a slowdown of 9.20% over all benchmarks (difference between **F** and **S**). The remaining slowdown is caused by the contract system itself. The subsequent detailed treatment of the score values splits the impact into its individual components.

Comparing columns **F** and **w/o C** shows that callback updates cause an overall slowdown of 4.25%. This point includes the recalculation of the callback constraints as explained in Section 3.7.

The numbers also show that decompiling functions has negligible impact on the execution time. Decompiling decreases the score by 6.29% over all benchmarks (compare columns **w/o C** and **w/o D**)[13]. This number is surprisingly low when taking into account that predicate evaluation includes recompiling all predicates on all runs as explained in Section 4.

Comparing the scores in columns **w/o D** and **w/o M** indicates that the membrane, as it is used by the sandbox, does not contribute significantly to the run-time overhead. It does not decrease the total scores.

Finally, after deactivating predicate execution, we see that pure predicate handling causes a slowdown of approximately 1.76% (this is the impact of the function calls). In contrast to column **S**, column **w/o P** shows the score values of the programs without sandboxing, without recompiling, and without callback updates, whereas in column **S** sandboxing, recompilation,

[13] Function recompilation can be safely deactivated for the benchmarks without changing the outcome because our generated base contracts are guaranteed to be free of side effects.

Benchmark	Contract			Sandbox		Callback
	A	I	P	M	D	
Richards	24	1599377224	935751200	4678756000	4	3351504000
DeltaBlue	54	2319477672	1340451212	6702256060	5	4744203248
Crypto	1	5	3	15	3	13
RayTrace	42	687240082	509234422	2546172110	4	2190186074
EarleyBoyer	3944	89022	68172	340860	6	309120
RegExp	0	0	0	0	0	0
Splay	10	11620663	7067593	35337965	5	26231845
SplayLatency	10	11620663	7067593	35337965	5	26231845
NavierStokes	51	48334	39109	195545	5	177197
pdf.js	3	15	9	45	4	39
Mandreel	7	57	28	140	4	128
MandreelLatency	7	57	28	140	4	128
Gameboy Emulator	3206	141669753	97487985	487439925	5	399084085
Code loading	5600	34800	18400	92000	4	70400
Box2DWeb	20075	172755100	112664947	563324735	5	469141435
zlib	0	0	0	0	0	0
TypeScript	4	12673644	8449090	42245450	2	33796350

Figure 3 Statistic from running the Google Octane 2.0 Benchmark Suite. Column **A** (*Assert*) shows the numbers of top-level contract assertions. Column **I** (*Internal*) contains the numbers of internal contract assertions whereby column **P** (*Predicate*) lists the number of predicate evaluations. Column **M** (*Membrane*) shows the numbers of wrap operation and column **D** (*Decompile*) show the numbers of decompile operations. The last column **Callback** gives the numbers of callback updates.

and callback updates remain active.

From the score values we find that the overall slowdown of sandboxed contract checking vs. a baseline without contracts amounts to a factor of 7136, approximately. The dramatic decrease of the score values in the heavily affected benchmarks is simply caused by the tremendous number of checks that arise during a run.

For example, in the *Splay* benchmark, the insert, find, and remove functions on trees are contracted. These functions are called every time a tree operation is performed. As the benchmark runs for 1400 iterations on trees with 8000 nodes, there is a considerable number of operations, each of which checks a contract. It should be recalled that every contract check performs at least two *typeof* checks.

Expressed in absolute time spans, contract checking causes a run time deterioration of 0.17ms for every single predicate check. For example, the contracted *Richards* requires 152480 seconds to complete and performs 935751200 predicate checks. Its baseline requires 8 seconds. Thus, contract checking requires 152472 seconds. That gives 0.16ms per predicate check.

Google claims that Octane "measure[s] the performance of JavaScript code found in large, real-world web applications, running on modern mobile and desktop browsers"[14]. For an academic, this is as realistic as it gets.

However, there are currently no large JavaScript application with contracts that we could use to benchmark, so we had to resort to automatic generation and insertion of contracts. These contracts may end up in artificial and unnatural places that would be avoided by an

[14] `https://developers.google.com/octane/`

efficiency conscious human developer. Thus, the numbers that we obtain give insight into the performance of our contract implementation, but they cannot be used to predict the performance impact of contracts on realistic programs with contracts inserted by human programmers. The scores of the real-world programs (*pdf.js*, *Mandreel*, *Code Loading*) among the benchmarks provide some initial evidence in this direction: their scores are much higher and they are only slightly effected by contract monitoring. But more experimentation is needed to draw a statistically valid conclusion.

6 Related Work

Contract Validation

Contracts may be validated statically or dynamically. Purely static frameworks (e.g. ESC/-Java [25]) transform specifications and programs into verification conditions to be verified by a theorem prover. Others [50, 45] rely on symbolic execution to prove adherence to contracts. However, most frameworks perform run-time monitoring as proposed in Meyer's work.

Higher-Order Contracts

Findler and Felleisen [24] first showed how to construct contracts and contract monitors for higher-order functional languages. Their work has attracted a plethora of follow-up works that range from semantic investigations [8, 23] over deliberations on blame assignment [15, 49] to extensions in various directions: contracts for polymorphic types [2, 7], for affine types [46], for behavioral and temporal conditions [17, 20], etc. While the cited semantic investigations consider noninterference, only Disney and coworkers [20] give noninterference a high priority and propose an implementation that enforces it. The other contract monitoring implementations that we are aware of, do not address noninterference or restrict their predicates.

Embedded Contract Language

Specification Languages like JML [38] state behavior in terms of a custom contract language or in terms of annotations in comments. An embedded contract language exploits the language itself to state contracts. Thus programmers need not learn a new language and the contract specification can use the full power of the language. Existing compilers and tools can be used without modifications.

Combinations of Contracts

Over time, a range of contract operators emanated, many of which are inspired by type operators. There are contract operators analogous to (dependent) function types [24], product types, sum types [33], as well as universal types [2]. Racket also implements restricted versions of conjunctions and disjunctions of contracts (see below). However, current systems do not support contracts analogous to union and intersection types nor do they support full boolean combination of contracts (negation is missing).

Dimoulas and Felleisen [14] propose a contract composition, which corresponds to a conjunction of contracts. But their operator is restricted to contracts of the same type. Before evaluating a conjunction it lifts the operator recursively to base contracts where it finally builds the conjunction of the predicate results.

Racket's contract system [26, Chapter 7] supports boolean combinations of contracts. Conjunctions of contracts are decomposed and applied sequentially [45]. Disjunctions of flat contracts are transformed so that the first disjunct does not cause a blame immediately if its predicate fails. However, Racket places severe restrictions on using disjunction with higher-order contracts and restricts negation to base contracts. A disjunction must be resolved by first-order choice to at most one higher-order contract; otherwise it is rejected at run time.

Proxies

The JavaScript proxy API [47] enables a developer to enhance the functionality of objects easily. JavaScript proxies have been used for Disney's JavaScript contract system, contracts.js [18], to enforce access permission contracts [35], as well as for other dynamic effects systems, meta-level extension, behavioral reflection, security, or concurrency control [40, 4, 9].

Sandboxing JavaScript

The most closely related work to our sandbox mechanism is the work of Arnaud et al. [3]. They provide features similar to our sandbox mechanism. Both approaches focus on access restriction and noninterference to guarantee side effect free assertions of contracts.

Our sandbox mechanism is inspired by the design of access control wrappers which is used for revocable references and membranes [47, 42]. In memory-safe languages, a function can only cause effects to objects outside itself if it holds a reference to the other object. The authority to affect the object can simply be removed if a membrane revokes the reference which detaches the proxy from its target.

Our sandbox works in a similar way and guarantees read-only access to target objects, but redirects write operations. Write access is completely forbidden and raises an exception. However, the restrictions affect only values that cross the border between the global execution environment and a predicate execution. Values that are defined and used in one side, e.g. local values, were not restricted. Write access to those values is fine.

Other approaches implement restrictions by filtering and rewriting untrusted code or by removing features that are either unsafe or that grant uncontrolled access. The Caja Compiler [28, 41], for example, compiles JavaScript code in a sanitized JavaScript subset that can safely be executed in normal engines. However, some static guarantees do not apply to code created at run time. For this reason Caja restricts dynamic features and adds run-time checks that prevent access to unsafe function and objects.

7 Conclusion

We presented *TreatJS*, a language embedded, dynamic, higher-order contract system for full JavaScript. *TreatJS* extends the standard abstractions for higher-order contracts with intersection and union contracts, boolean combinations of contracts, and parameterized contracts, which are the building blocks for contracts that depend on run-time values. *TreatJS* implements proxy-based sandboxing for all code fragments in contracts to guarantee that contract evaluation does not interfere with normal program execution. The only serious impediment to full noninterference lies in JavaScript's treatment of proxy equality, which considers a proxy as an individual object.

The impact of contracts on the execution time varies widely depending on the particular functions that are under contract and on the frequency with which the functions are called.

While some programs' run time is heavily impacted, others are nearly unaffected. We believe that if contracts are carefully and manually inserted with the purpose of determining interface specifications and finding bugs in a program, their run time will mostly be unaffected. But more experimentation is needed to draw a statistically valid conclusion.

References

1. Parker Abercrombie and Murat Karaorman. jContractor: Design by contract for Java. `http://jcontractor.sourceforge.net/`, 2003.
2. Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In Thomas Ball and Mooly Sagiv, editors, *Proceedings 38th Annual ACM Symposium on Principles of Programming Languages*, pages 201–214, Austin, TX, USA, January 2011. ACM Press.
3. Jean-Baptiste Arnaud, Marcus Denker, Stéphane Ducasse, Damien Pollet, Alexandre Bergel, and Mathieu Suen. Read-only execution for dynamic languages. In Jan Vitek, editor, *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, volume 6141 of *Lecture Notes in Computer Science*, pages 117–136, Málaga, Spain, June 2010. Springer.
4. Thomas H. Austin, Tim Disney, and Cormac Flanagan. Virtual values for language extension. In Cristina Videira Lopes and Kathleen Fisher, editors, *OOPSLA*, pages 921–938, Portland, OR, USA, 2011. ACM.
5. Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de' Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.
6. Nuel Belnap. A useful four-valued logic. In J. Michael Dunn and George Epstein, editors, *Modern Uses of Multiple-Valued Logic*, volume 2 of *Episteme*, pages 5–37. Springer Netherlands, 1977.
7. João Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C. Pierce. Polymorphic contracts. In Gilles Barthe, editor, *Proceedings of the 20th European Symposium on Programming*, volume 6602 of *Lecture Notes in Computer Science*, pages 18–37, Saarbrücken, Germany, March 2011. Springer-Verlag.
8. Matthias Blume and David McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16:375–414, July 2006.
9. Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA*, pages 331–344. ACM, 2004.
10. Lorenzo Caminiti. Contract++, contract programming library for C++. `http://sourceforge.net/projects/contractpp/`, August 2012.
11. Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for Web services. *ACM Transactions on Programming Languages and Systems*, 31(5), 2009.
12. Olaf Chitil. Practical typed lazy contracts. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*. ACM, September 2012.
13. Mario Coppo and M. Dezani-Ciancaglini. A new type-assignment for λ-terms. *Archiv. Math. Logik*, 19(139-156), 1978.
14. Christos Dimoulas and Matthias Felleisen. On contract satisfaction in a higher-order world. *ACM Transactions on Programming Languages and Systems*, 33(5):16, 2011.
15. Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: No more scapegoating. In Thomas Ball and Mooly Sagiv, editors, *Proceedings 38th Annual ACM Symposium on Principles of Programming Languages*, pages 215–226, Austin, TX, USA, January 2011. ACM Press.
16. Christos Dimoulas, Riccardo Pucella, and Matthias Felleisen. Future contracts. In António Porto and Francisco J. López-Fraguas, editors, *Principles and Practice of Declarative Programming, PPDP 2009*, pages 195–206, Coimbra, Portugal, September 2009. ACM.

17 Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *European Symposium on Programming*, volume 7211 of *Lecture Notes in Computer Science*, Tallinn, Estland, April 2012. Springer-Verlag.

18 Tim Disney. contracts.js. `https://github.com/disnet/contracts.js`, April 2013.

19 Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. Sweeten your JavaScript: Hygienic macros for ES5. In Andrew P. Black and Laurence Tratt, editors, *DLS*, pages 35–44, Portland, OR, USA, October 2014. ACM.

20 Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal higher-order contracts. In Olivier Danvy, editor, *Proceedings International Conference on Functional Programming 2011*, pages 176–188, Tokyo, Japan, September 2011. ACM Press, New York.

21 ECMAScript Language Specification, December 2009. ECMA International, ECMA-262, 5th edition.

22 Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2103–2110, Sierre, Switzerland, 2010. ACM.

23 Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. In Philip Wadler and Masami Hagiya, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming FLOPS 2006*, pages 226–241, Fuji Susono, Japan, April 2006. Springer-Verlag.

24 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In Simon Peyton-Jones, editor, *Proceedings International Conference on Functional Programming 2002*, pages 48–59, Pittsburgh, PA, USA, October 2002. ACM Press, New York.

25 Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 2002. ACM Press.

26 Matthew Flatt, Robert Bruce Findler, and PLT. *The Racket Guide*, v.6.0 edition, March 2014. `http://docs.racket-lang.org/guide/index.html`.

27 Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

28 google-caja: A source-to-source translator for securing JavaScript-based web content. `http://code.google.com/p/google-caja/`, (as of 2011).

29 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Proceedings 37th Annual ACM Symposium on Principles of Programming Languages*, pages 353–364, Madrid, Spain, January 2010. ACM Press.

30 David R. Hanson and Todd A. Proebsting. Dynamic variables. In *Proceedings of the 2001 Conference on Programming Language Design and Implementation*, pages 264–273, Snowbird, UT, USA, June 2001. ACM Press, New York, USA.

31 Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. Access permission contracts for scripting languages. In John Field and Michael Hicks, editors, *Proceedings 39th Annual ACM Symposium on Principles of Programming Languages*, pages 111–122, Philadelphia, USA, January 2012. ACM Press.

32 Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In Norman K. Meyrowitz, editor, *Conference Object Oriented Programming, Systems, Languages, and Applications / European Conference on Object-Oriented Programming*, pages 169–180, Ottawa, Canada, October 1990.

33 Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In Philip Wadler and Masami Hagiya, editors, *Proceedings of the 8th International Sym-*

posium on Functional and Logic Programming FLOPS 2006, pages 208–225, Fuji Susono, Japan, April 2006. Springer-Verlag.

34 Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent object proxies for JavaScript. In John Boyland, editor, *ECOOP*, volume ?, Prague, Czech Repulic, July 2015. LIPICS. to appear.

35 Matthias Keil and Peter Thiemann. Efficient dynamic access analysis using JavaScript proxies. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 49–60, New York, NY, USA, 2013. ACM.

36 Matthias Keil and Peter Thiemann. TreatJS: Higher-order contracts for JavaScript. Technical report, Institute for Computer Science, University of Freiburg, 2015.

37 Reto Kramer. iContract — the Java design by contract tool. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 295–307, Santa Barbara, CA, USA, 1998. IEEE Computer Society.

38 Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188, Norwell, MA, USA, 1999. Kluwer Academic Publishers.

39 Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

40 Mark S. Miller, Tom Van Cutsem, and Bill Tulloh. Distributed electronic rights in JavaScript. In Matthias Felleisen and Philippa Gardner, editors, *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 1–20, Rome, Italy, March 2013. Springer.

41 Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized JavaScript. `http://google-caja.googlecode.com`, 2008. Google White Paper.

42 Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006. AAI3245526.

43 John C. Reynolds. Gedanken - a simple typeless language based on the principle of completeness and the reference concept. *Commun. ACM*, 13(5):308–319, 1970.

44 T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: Run-time support for reasonable interposition. In Gary T. Leavens and Matthew B. Dwyer, editors, *OOPSLA*, pages 943–962. ACM, 2012.

45 Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In Gary T. Leavens and Matthew B. Dwyer, editors, *OOPSLA*, pages 537–554. ACM, 2012.

46 Jesse A. Tov and Riccardo Pucella. Stateful contracts for affine types. In Andrew D. Gordon, editor, *ESOP 2010*, volume 6012 of *Lecture Notes in Computer Science*, pages 550–569. Springer-Verlag, 2010.

47 Tom Van Cutsem and Mark S. Miller. Proxies: Design principles for robust object-oriented intercession APIs. In William D. Clinger, editor, *DLS*, pages 59–72. ACM, 2010.

48 Tom Van Cutsem and Mark S. Miller. Trustworthy proxies - virtualizing objects with invariants. In Giuseppe Castagna, editor, *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 154–178, Montpellier, France, July 2013. Springer.

49 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proceedings 18th European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16, York, UK, March 2009. Springer.

50 Dana N. Xu, Simon L. Peyton Jones, and Koen Claessen. Static contract checking for Haskell. In Benjamin Pierce, editor, *Proceedings 36th Annual ACM Symposium on Principles of Programming Languages*, pages 41–52, Savannah, GA, USA, January 2009. ACM Press.

Trust, but Verify: Two-Phase Typing for Dynamic Languages

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala

Univeristy of California, San Diego
La Jolla, CA, 92093, USA
{pvekris,blcosman,rjhala}@cs.ucsd.edu

Abstract

A key challenge when statically typing so-called dynamic languages is the ubiquity of *value-based overloading*, where a given function can dynamically reflect upon and behave according to the types of its arguments. Thus, to establish basic types, the analysis must reason precisely about values, but in the presence of higher-order functions and polymorphism, this reasoning itself can require basic types. In this paper we address this chicken-and-egg problem by introducing the framework of two-phased typing. The first "trust" phase performs classical, i.e. flow-, path- and value-insensitive type checking to assign basic types to various program expressions. When the check inevitably runs into "errors" due to value-insensitivity, it wraps problematic expressions with DEAD-casts, which explicate the trust obligations that must be discharged by the second phase. The second phase uses refinement typing, a flow- and path-sensitive analysis, that decorates the first phase's types with logical predicates to track value relationships and thereby verify the casts and establish other correctness properties for dynamically typed languages.

1998 ACM Subject Classification D.3.3 [Programming Languages] Language Constructs and Features – Constraints, Polymorhpism, F.3.1 [Logics and Meanings of Programs] Specifying and Verifying and Reasoning about Programs – Assertions, Pre- and post-conditions, F.3.3 [Logics and Meanings of Programs] Studies of Program Constructs – Type structure

Keywords and phrases Dynamic Languages, Type Systems, Refinement Types, Intersection Types, Overloading

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.52

1 Introduction

Higher-order constructs are increasingly adopted in *dynamic scripting* languages, as they facilitate the production of clean, correct and maintainable code. Consider, for example, the following (first-order) JavaScript function

```
function minIndexFO(a) {
  if (a.length ≤ 0)
    return -1;
  var min = 0;
  for (var i = 0; i < a.length; i++) {
    if (a[i] < a[min])
      min = i;
  }
  return min;
}
```

which computes the index of the minimum value in the array a by looping over the array, updating the min value with each index i whose value a[i] is smaller than the "current" a[min]. Modern dynamic languages let programmers factor the looping pattern into a

© Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 52–75

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
function $reduce(a, f, x) {
  var res = x;
  for (var i = 0; i < a.length; i
      ++)
    res = f(res, a[i], i);
  return res;
}

function reduce(a, f, x) {
  if (arguments.length === 3)
    return $reduce(a, f, x);
  return $reduce(a, f, a[0]);
}
```

```
function minIndex(a) {
  if (a.length ≤ 0)
    return -1;
  function step(min, cur, i) {
    return cur < a[min] ? i:min;
  }
  return reduce(a, step, 0);
}
```

Figure 1 Computing the minimum-valued index with Higher-Order Functions.

higher-order `$reduce` function (Figure 1), which frees them from manipulating indices and thereby prevents the attendant "off-by-one" mistakes. Instead, the programmer can compute the minimum index by supplying an appropriate `f` to `reduce` as in `minIndex` shown at the right of Figure 1.

This trend towards abstraction and reuse poses a challenge to static program analyses: *how to precisely trace value relationships across higher-order functions and containers?* A variety of dataflow- or abstract interpretation- based analyses could be used to verify the safety of array accesses in `minIndexFO` by inferring the loop invariant that `i` and `min` are between `0` and `a.length`. Alas, these analyses would fail on `minIndex`. The usual methods of procedure summarization apply to first-order functions, and it is not clear how to extend higher-order analyses like CFA to track the *relationships* between the values and closures that flow to `$reduce`.

An Approach: Refinement Types. Refinement types [31] hold the promise of a precise and compositional analysis for higher-order functions. Here, *basic* types are decorated with *refinement* predicates that constrain the values inhabiting the type. For example, we can define

```
type idx<x> = {v:number | 0 ≤ v && v < len(x) }
```

to denote the set of valid indices for an array `x` and can be used to type `$reduce` as

```
$reduce :: <A,B>(a: A[], f: (B,A,idx<a>) ⇒ B, x: B) ⇒ B
```

The above type is a precise *relational summary* of the behavior of `$reduce`: the higher-order `f` is only invoked with valid indices for `a`. Consequently, `step` is only called with valid indices for `a`, which ensures array safety.

Problem: Value-based Overloading. A main attraction of dynamic languages is *value-based overloading*, where syntactic entities (*e.g.* variables) may be bound to multiple types at run-time, and furthermore, computations may be customized to particular types, by reflecting on the values bound to variables. For example, it is common to simplify APIs by overloading the `reduce` function to make the initial value `x` optional; when omitted, the first array element `a[0]` is used instead (Figure 1). Here, `reduce` really has *two* different function types: one with 3 parameters and another one with 2. Furthermore, `reduce` *reflects* on the size of `arguments` to select the behavior appropriate to the calling context.

Value-based overloading conflicts with a crucial prerequisite for refinements, namely that the language possesses an *unrefined* static type system that provides basic invariants about

values which can then be refined using logical predicates. Unfortunately, as shown by `reduce`, to soundly establish basic typing we must reason about the logical relationships between values, which is exactly the problem we wished to solve via refinement typing. In other words, value-based overloading creates a chicken-and-egg problem: refinements require us to first establish basic typing, but the latter itself requires reasoning about values (and hence, refinements!).

Solution: Trust but Verify. We introduce *two-phased typing*, a new strategy for statically analyzing dynamic languages. The key insight is that we can completely decouple reasoning about *basic* types and *refinements* into distinct phases by converting "type errors" from the first phase into "assertion failures" for the second. Two-phase typing starts with a source language where value-based overloading is specified using *intersections* and (untagged) *unions* of the different possible (run-time) types.

The first phase performs classical, *i.e.* flow-, path- and value-insensitive type checking to assign basic types to various program expressions. When the check inevitably runs into "errors" due to value-insensitivity, it wraps problematic expressions with `DEAD`-casts which allow the first phase to proceed, *trusting* that the expressions have the casted types. In other words, the first phase *elaborates* [10] the source language with intersection and (untagged) union types, into a target ML-like language with classical products, (tagged) sums and `DEAD`-casts, which explicate the trust obligations that must be discharged by the second phase. The second phase carries out *refinement*, *i.e.* flow- and path-sensitive inference, to decorate the basic types (from the first phase) with predicates that precisely track relationships about values, and uses the refinements to *verify* the casts and other properties, discharging the assumptions of the first phase.

For example, `reduce` is described as the intersection of two contexts, *i.e.* function types which take two and three parameters respectively. The trust-phase checks the body under both contexts (separately). In each context, one of the calls to `$reduce` is "ill-typed". In the context where the function takes two inputs, the call using `x` is undefined; when the function takes three inputs, there is a mismatch in the types of `f` and `a[0]`. Consequently, each ill-typed expression is wrapped with a *cast* which obliges the verify phase to prove that the call is dead code in that context, thereby verifying overloading in a cooperative manner.

Benefits. While it is possible to account for value-based overloading in a single phase, the currently known methods that do so are limited to the extremes of types and program logics. At one end, systems like Typed Racket [28] and Flow Typing [16] extend classical type systems to account for a fixed set of `typeof`-style tests, but cannot reason about general value tests (*e.g.* the size of `arguments`) that often appear in idiomatic code. At the other end, systems like System D [7] embed the typing relation in an expressive program logic, allowing general value tests, but give up on basic type structure, thereby sacrificing inference, causing a significant annotation overhead. In contrast, our approach separates the concerns of basic typing and reasoning about values, thereby yielding several concrete benefits by *modularizing* specification, verification and soundness.

- **Specification:** Instead of a fixed set of type-tests, two-phase typing handles complex value relationships which can be captured inside refinements in an expressive logic. Furthermore, the *expressiveness* of the basic type system and logics can be extended independently, *e.g.* to account for polymorphism, classes or new logical theories, directly yielding a more expressive specification mechanism.

```
neg :: (number, number) ⇒ number          var a = neg(1,1);     // OK
     ∧ (number, boolean) ⇒ boolean         var b = neg(0,true); // OK
function neg(flag, x) {                    var c = neg(0,1);     // ERR
  if (flag) return 0-x;                    var d = neg(1,true); // ERR
  return !x;
}
```

Figure 2 An example program with value-based overloading.

- **Verification:** Two-phase typing enables the straightforward composition of simple type checkers (uncomplicated by reasoning about values) with program logics (relying upon the basic invariants provided by typing – *e.g.* the parametric polymorphism needed to verify `minIndex`). Furthermore, two-phase typing allows us to compose basic typing with abstract interpretation [23], which drastically lowers the annotation burden for using refinement types.
- **Soundness:** Finally, our elaboration-based approach makes it straightforward to establish soundness for two-phased typing. The first phase ignores values and refinements, so we can use classical methods to prove the elaborated target is "equivalent to" the source. The second phase uses standard refinement typing techniques on the well-typed elaborated target, and hence lets us directly reuse the soundness theorems for such systems [18] to obtain end-to-end soundness for two-phased typing.

Contributions. Concretely, in this paper we make the following contributions. First, we informally illustrate (Section 2) how two-phase typing lets us statically analyze dynamic, value-based overloading patterns drawn from real-world code, where, we empirically demonstrate, value-based overloading is ubiquitous. Second, we formalize two-phase typing using a core calculus, Rsc, whose syntax and semantics are detailed in Section 3. Third, we formalize the first phase (Section 4), which *elaborates* [10] a source language with value-based overloading into a target language with `DEAD`-casts in lieu of overloading. We prove that the elaborated target preserves the semantics of the source, *i.e.* the `DEAD`-casts fail iff the source would hit a type error at run time. Finally, we demonstrate how standard refinement typing machinery can be applied to the elaborated well-typed target (Section 5) to statically verify the `DEAD`-casts, yielding end-to-end soundness for our system.

2 Overview

We begin with an overview illustrating how we soundly verify value-based overloading using our novel two-phased approach.

2.1 Value-based Overloading

Consider the code in Figure 2. The function `neg` behaves as follows. When a `number` is passed as input, indicated by passing in a *non-zero*, *i.e.* "truthy" `flag`, the function flips its sign by subtracting the input from `0`. Instead, when a `boolean` is passed in, indicated by a *zero*, *i.e.* "falsy" `flag`, the function returns the boolean negation. Hence, the calls made to assign `a` and `b` are legitimate and should be statically accepted. However, the calls made to assign `c` and `d` lead to run-time errors (assuming we eschew implicit coercions), and hence, should be rejected.

The function `neg` distils value-based overloading to its essence: a run-time test on one parameter's value is used to determine the type of, and hence the operation to be applied to,

File	#Funs	%Ovl	%Opt	%Any
box2d	529	0	3	**3**
ace	484	1	5	**6**
pixi	123	0	12	**12**
fabricjs	371	5	9	**13**
threejs	1022	1	24	**24**
leaflet	414	12	38	**41**
underscore	344	25	34	**45**
sugar	446	29	37	**48**
d3	475	43	17	**52**
jquery	226	52	31	**67**

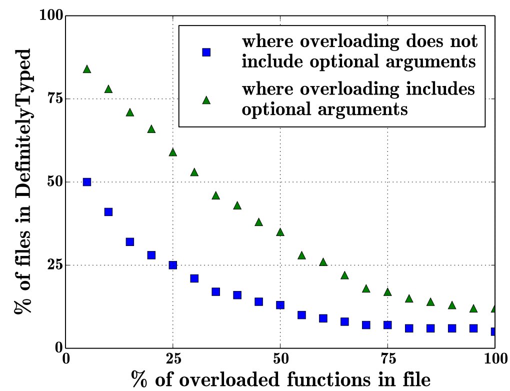


Figure 3 The prevalence of value-based overloading. **(L)** Libraries from the survey of Feldthaus *et al.* [12]: **#Funs** is the number of functions in the signature, **%Ovl** is %-functions with *multiple* signatures, **%Opt** is %-functions with *optional* arguments, and **%Any** is %-functions with either of these features. **(R)** Overloading across *all* files in Definitely Typed. A point (x, y) means y% of files have more than x% overloaded functions.

another value. Of course in JavaScript, one could use a single parameter and the `typeof` operator for this particular simple case, and design analyses targeted towards a fixed set of type tests, *e.g.* using variants of the `typeof` operator [28, 16]. However, arbitrary value tests – such as tests of the size of `arguments` shown in `reduce` in Figure 1 – can be and are used in practice. Thus, we illustrate the generality of the problem and our solution *without* using the `typeof` operator (which is a special case of our solution).

Prevalence of Value-based Overloading. The code from Figure 1 is not a pathological toy example. It is adapted from the widely used D3 visualization library. The advent of TypeScript makes it possible to establish the prevalence of value-based overloading in real-world libraries, as it allows developers to specify overloaded signatures for functions. (Even though TypeScript does not verify those signatures, it uses them as trusted interfaces for external JavaScript libraries and code completion.) The Definitely Typed repository [1] contains TypeScript interfaces for a large number of popular JavaScript libraries. We analyzed the TypeScript interfaces to determine the prevalence of value-based overloading. Intuitively, every function or method with multiple (overloaded) signatures or optional arguments has an implementation that uses value-based overloading.

Figure 3 summarizes the results of our study. On the left, we show the fraction of overloaded functions in the 10 benchmarks analyzed by Feldthaus *et al.* [12]. The data shows that over 25% of the functions in 4 of 10 libraries use value-based overloading, and an even larger fraction is overloaded in libraries like `jquery` and `d3`. On the right we summarize the occurrence of overloading across all the libraries in Definitely Typed. The data shows, for example, that in more than 25% of the libraries, *more than* 25% of the functions are overloaded with multiple types. The figure jumps to nearly 55% of functions if we also include optional arguments.

The signatures in Definitely Typed have not been soundly checked against[2] their implementations. Hence, it is possible that they mischaracterize the semantics of the actual code,

[1] `http://definitelytyped.org`
[2] Feldthaus *et al.* [12] describe an effective but unsound inconsistency detector.

but modulo this caveat, we believe the study demonstrates that value-based overloading is ubiquitous, and so to soundly and statically analyze dynamic languages, it is crucial that we develop techniques that can precisely and flexibly account for it.

2.2 Refinement Types

Types and Refinements. A basic refinement type T is a basic type, *e.g.* `number`, refined with a logical formula from an SMT decidable logic – for the purposes of this paper, the quantifier-free logic of uninterpreted functions and linear integer arithmetic (QF_UFLIA [25]). For example, `{v:number | v != 0}` describes the *subset* of numbers that are non-zero. We write A to abbreviate the trivially refined type $\{\nu : A \mid \mathit{true}\}$, *e.g.* `number` is an abbreviation for $\{\nu : \texttt{number} \mid \mathit{true}\}$.

Summaries: Function Types. We can specify the behavior of functions with refined function types, of the form

$$(x_1 : T_1, \ldots, x_n : T_n) \Rightarrow T$$

where arguments are named x_i and have types T_i and the output is a T. In essence, the *input* types T_i specify the function's preconditions, and the *output* type T describes the postcondition. Furthermore, each input type and the output type can *refer to* the arguments x_i which yields precise function contracts. For example,

$$(x : 0 \leq x) \Rightarrow \{\nu : \texttt{number} \mid x < \nu\}$$

is a function type that describes functions that *require* a non-negative input, and *ensure* that the output is greater than the input.

Example. Returning to `neg` in Figure 2, we can define two refinements of `number`:

```
type tt = {v:number | v != 0}     // "truthy" numbers
type ff = {v:number | v  = 0}     // "falsy"  numbers
```

which are used to specify a refined type for `neg` shown on the left in Figure 4.

Problem: A Circular Dependency. While it is easy enough to specify a type signature, it is another matter to verify it, and yet another matter to ensure soundness. The challenge is that value-based overloading introduces a circular dependency between types and refinements. The soundness of basic types requires (*i.e.* is established by) the refinements, while the refinements themselves require (*i.e.* are attached to) basic types. In classical refinement systems like DML [31], basic types are established *without* requiring refinements. A classical refinement system is thus a conservative extension of the corresponding non-refined language, *i.e.* removing the refinements from a DML program, yields valid, well-typed ML. Unfortunately, value-based overloading removes this crucial property, posing a circular dependency between types and refinements.

Solution: Two-Phase Checking. We break the cycle by typing programs in two phases. In the first, we *trust* the basic types are correct and use them (ignoring the refinements) to elaborate source programs into a target overloading-free language. Inevitably, value-based overloading leads to "errors" when typing certain sub-expressions in the wrong context, *e.g.* subtracting a `boolean`-valued `x` from `0`. Instead of rejecting the program, the elaboration wraps ill-typed expressions with `DEAD`-casts, which are assertions stating the program is

```
neg :: (tt, number) ⇒ number
    ∧ (ff, boolean) ⇒ boolean
function neg(flag, x) {
  if (flag) return 0-x;
  return !x;
}

var a = neg(1,1);    //OK
var b = neg(0,true); //OK
var c = neg(0,1);    //ERR
var d = neg(1,true); //ERR
```

```
neg#1 :: (tt, number) ⇒ number
function neg#1(flag, x) {
  if (flag) return 0-x;
  return !DEAD(x);
}
neg#2 :: (ff, boolean) ⇒ boolean
function neg#2(flag, x) {
  if (flag) return 0-DEAD(x);
  return !x;
}
var neg = (neg#1, neg#2);

var a = fst(neg)(1,1);    //OK
var b = snd(neg)(0,true); //OK
var c = fst(neg)(0,1);    //ERR
var d = snd(neg)(1,true); //ERR
```

Figure 4 Source program (l) and target (r) resulting from first phase elaboration.

well-typed *assuming* those expressions are dead code. In the second phase we reuse classical refinement typing techniques to *verify* that the `DEAD`-casts are indeed unreachable, thereby discharging the assumptions made in the first phase.

2.3 Phase 1: Trust

The first phase *elaborates* the source program into an equivalent typed target language with two key properties: First, the target program is simply typed – *i.e.* has *no* union or intersection types, but just classical ML-style sums and products. Second, source-level type errors are elaborated to target-level `DEAD`-casts. The right side of Figure 4 shows the elaboration of the source from the left side. While we formalize the elaboration declaratively using a single judgment form (Section 4), it comprises two different steps. Critically, each step, and hence the entire first phase, is *independent* of the refinements – they are simply carried along unchanged.

A. Clone. In the first step, we create separate clones of each overloaded function, where each clone is assigned a single conjunct of the original overloaded type. For example, we create two clones `neg#1` and `neg#2` respectively typed using the two conjuncts of the original `neg`. The binder `neg` is replaced with a *tuple* of its clones. Finally, each use of `neg` extracts the appropriate element from the tuple before issuing the call.

Since the trust phase must be independent of refinements, the overload resolution in this step uses *only* the basic types at the call-site to determine which of the two clones to invoke. For example, in the assignment to `a`, the source call `neg(1,1)` – which passes in two `number` values, and hence, matches the first overload (conjunct) – is elaborated to the target call `fst(neg)(1,1)`. In the assignment to `d`, the source call `neg(1,true)` – which passes in a `number` and a `boolean`, and hence matches the second overload – is elaborated to the target call `snd(neg)(1,true)`, even though `1` does *not* have the refined type `ff`.

B: Cast. In the second step we check – using classical, unrefined type checking – that each clone adheres to its specified type. Unlike under usual intersection typing [22, 10], in our context these checks almost surely "fail". For example, `neg#1` *does not* type-check as the parameter `x` has type `number` and so we cannot compute `!x`. Similarly, `neg#2` fails because `x` has type `boolean` and so `0-x` is erroneous. Rather than reject the program, we wrap such

failures with DEAD-casts. For example, the above occurrences of `x` elaborate to DEAD(`x`) on the right in Figure 4.

Intuitively, the *value relationships* established at the call-sites and guards ensure that the failures will not happen at run-time. However, recall that the first phase's goal is to decouple reasoning about types from reasoning about values. Hence, we just *trust* all the types but use DEAD-casts to *explicate* the value-relationship obligations that are needed to establish typing: namely that the DEAD-casts are indeed dead code.

2.4 Phase 2: Verify

The second phase takes as input the elaborated program emitted by the first phase, which is essentially a classical *well-typed* ML program with assertions and without any value-overloading. Hence, the second phase can use any existing program logic [14, 4], refinement typing [31, 18, 23, 2], or contracts & abstract interpretation [20] to check that the target's assertions never fail, which, we prove, ensures that the source is type-safe.

To analyze programs with closures, collections and polymorphism, (*e.g.* `minIndex` from Figure 1) we perform the second phase using the refinement types that are carried over unchanged by the elaboration process of the first phase. Intuitively, refinement typing can be viewed as a generalization of classical program logics where *assertions* are generalized to type bindings, and the rule of *consequence* is generalized as subtyping. While refinement typing is a previously known technique, to make the paper self-contained, we illustrate how the second phase verifies the DEAD-casts in Figure 4.

Refinement Type Checking. A refinement type checker works by building up an *environment* of type bindings that describe the machine state at each program point, and by checking that at each call-site, the actual argument's type is a refined *subtype* of the expected type for the callee, under the context described by the environment at that site. The subtyping relation for basic types is converted to a logical *verification condition* whose validity is checked by an SMT solver. The subtyping relation for *compound* types (*e.g.* functions, collections) is decomposed, via co- and contra-variant subtyping rules, into subtyping constraints over *basic* types, which can be discharged as above.

Typing DEAD-Casts. To use a standard refinement type checker for the second phase of verification, we only need to treat DEAD as a primitive operation with the refined type:

$$\texttt{DEAD} :: \forall A, B.(\{\nu : A \mid \mathit{false}\}) \Rightarrow B$$

That is, we assign DEAD the *precondition false* which states there are *no* valid inputs for it, *i.e.* that it should never be called (akin to `assert(false)` in other settings).

Environments. To verify DEAD-casts, the refinement type checker builds up an environment of type binders describing *variables* and *branch conditions* that are in scope at each program point. For example, the DEAD call in `neg#1`, has the environment:

$$\Gamma_1 \doteq \texttt{flag}{:}\texttt{tt},\ \texttt{x}{:}\texttt{number},\ \texttt{g}_1{:}\{\nu{:}\texttt{boolean} \mid \texttt{flag} = 0\} \tag{1}$$

where the first two bindings are the function parameters, whose types are the input types. The third binding is from the "else" branch of the `flag` test, asserting the branch condition `flag` is "falsy" *i.e.* equals 0. At the DEAD call in `neg#2` the environment is:

$$\Gamma_2 \doteq \texttt{flag}{:}\texttt{ff},\ \texttt{x}{:}\texttt{boolean},\ \texttt{g}_1{:}\{\nu{:}\texttt{boolean} \mid \texttt{flag} \neq 0\} \tag{2}$$

At the assignments to `a`, `b` and `c` the environments are respectively:

$$\Gamma_a \doteq \texttt{neg}{:}T_{\texttt{neg}} \tag{3}$$
$$\Gamma_b \doteq \Gamma_a,\ \texttt{a}{:}\texttt{number} \tag{4}$$
$$\Gamma_c \doteq \Gamma_b,\ \texttt{b}{:}\texttt{boolean} \tag{5}$$

where $T_{\texttt{neg}}$ abbreviates the *product* type of the (elaborated) tuple `neg`.

$$T_{\texttt{neg}} \doteq ((\texttt{tt}, \texttt{number}) \Rightarrow \texttt{number}) \times ((\texttt{ff}, \texttt{boolean}) \Rightarrow \texttt{boolean}) \tag{6}$$

Subtyping. At each function call-site, the refinement type system checks that the *actual* argument is indeed a subtype of the *expected* one. For example, the `DEAD` calls inside `neg#1` and `neg#2` yield the respective subtyping obligation:

$$\Gamma_1 \vdash \{\nu{:}\texttt{number} \mid \nu = \texttt{x}\} \sqsubseteq \{\nu{:}\texttt{number} \mid \mathit{false}\} \tag{7}$$
$$\Gamma_2 \vdash \{\nu{:}\texttt{boolean} \mid \nu = \texttt{x}\} \sqsubseteq \{\nu{:}\texttt{boolean} \mid \mathit{false}\} \tag{8}$$

The obligation states that the type of the argument `x` should be a subtype of the input type of `DEAD`. Similarly, at the assignments to `a`, `b` and `c` the first arguments generate the respective subtyping obligations:

$$\Gamma_a \vdash \{\nu{:}\texttt{number} \mid \nu = 1\} \sqsubseteq \{\nu{:}\texttt{number} \mid \nu \neq 0\} \tag{9}$$
$$\Gamma_b \vdash \{\nu{:}\texttt{number} \mid \nu = 0\} \sqsubseteq \{\nu{:}\texttt{number} \mid \nu = 0\} \tag{10}$$
$$\Gamma_c \vdash \{\nu{:}\texttt{number} \mid \nu = 0\} \sqsubseteq \{\nu{:}\texttt{number} \mid \nu \neq 0\} \tag{11}$$

Verification Conditions. To verify subtyping obligations, we convert them into logical verification conditions (VCs), whose validity determines whether the subtyping holds. A subtyping obligation $\Gamma \vdash \{\nu{:}b \mid p\} \sqsubseteq \{\nu{:}b \mid q\}$ translates to the VC $[\![\Gamma]\!] \Rightarrow (p \Rightarrow q)$ where $[\![\Gamma]\!]$ is the conjunction of the refinements of the binders in Γ. For example, the subtyping obligations (7) and (8) yield the respective VCs:

$$(\texttt{flag} \neq 0 \wedge \mathit{true} \wedge \texttt{flag} = 0) \Rightarrow \nu = \texttt{x} \Rightarrow \mathit{false} \tag{12}$$
$$(\texttt{flag} = 0 \wedge \mathit{true} \wedge \texttt{flag} \neq 0) \Rightarrow \nu = \texttt{x} \Rightarrow \mathit{false} \tag{13}$$

Here, the conjunct *true* arises from the trivial refinements *e.g.* the binding for `x`. The above VCs are deemed valid by an SMT solver as the hypotheses are inconsistent, which proves the call is indeed dead code. Similarly, (9), (10) respectively yield VCs:

$$\mathit{true} \Rightarrow \nu = 1 \Rightarrow \nu \neq 0 \tag{14}$$
$$\mathit{true} \Rightarrow \nu = 0 \Rightarrow \nu = 0 \tag{15}$$

which are deemed valid by SMT, verifying the assignments to `a`, `b`. However, by (11):

$$\mathit{true} \Rightarrow \nu = 0 \Rightarrow \nu \neq 0 \tag{16}$$

which is invalid, ensuring that we *reject* the call that assigns to `c`.

2.5 Two-Phase Inference

Our two-phased approach readily lends itself to abstract interpretation based *refinement inference* which can drastically lower the programmer annotations required to verify various

safety properties, *e.g.* reducing the annotations needed to verify array bounds safety in ML programs from 31% of code size to under 1% [23]. Here we illustrate how inference works in the presence of value-based overloading. Suppose we are *not* given the refinements for the signature of `neg` but only the unrefined signature (either given to us explicitly as in TypeScript, or inferred via dataflow analysis [16, 11]). As inference is difficult with incorrect code, we omit the erroneous statements that assign to `c` and `d`.

Refinement inference proceeds in three steps. First, we create *templates* which are the basic types decorated with *refinement variables* κ in place of the unknown refinements. Second, we perform the *trust* phase to elaborate the source program into a well-typed target free of overloading. Remember that this phase uses only the basic types and is oblivious to the (in this case unknown) refinements. Third, we perform the *verify* phase which now generates VCs over the refinement variables κ. These VCs – *logical implications* between the refinements and κ variables – correspond to so-called Horn constraints over the κ variables, and can be solved via abstract interpretation [13, 23].

0. Templates: Let us revisit the program from Figure 2, with the goal of inferring the refinements. Recall that the (unrefined) type of `neg` is:

$$\begin{aligned}\texttt{neg} :: &\ (\texttt{number}, \texttt{number}) \Rightarrow \texttt{number}\\ &\wedge (\texttt{number}, \texttt{boolean}) \Rightarrow \texttt{boolean}\end{aligned}$$

We create a *template* by refining each base type with a (distinct) refinement variable:

$$\begin{aligned}\texttt{neg} :: &\ (\{\nu{:}\texttt{number} \mid \kappa_1\}, \{\nu{:}\texttt{number} \mid \kappa_2\}) \Rightarrow \{\nu{:}\texttt{number} \mid \kappa_3\}\\ &\wedge (\{\nu{:}\texttt{number} \mid \kappa_4\}, \{\nu{:}\texttt{boolean} \mid \kappa_5\}) \Rightarrow \{\nu{:}\texttt{boolean} \mid \kappa_6\}\end{aligned}$$

1. Trust: The trust phase proceeds as before, propagating the refinements to the signatures of the elaborated target, yielding the code on the right in Figure 4 except that `neg#1` and `neg#2` have the respective templates:

$$\texttt{neg\#1} :: (\{\nu{:}\texttt{number} \mid \kappa_1\}, \{\nu{:}\texttt{number} \mid \kappa_2\}) \Rightarrow \{\nu{:}\texttt{number} \mid \kappa_3\}$$
$$\texttt{neg\#2} :: (\{\nu{:}\texttt{number} \mid \kappa_4\}, \{\nu{:}\texttt{boolean} \mid \kappa_5\}) \Rightarrow \{\nu{:}\texttt{boolean} \mid \kappa_6\}$$

2. Verify: The verify phase proceeds as before, but using templates instead of the types. Hence, at the `DEAD`-cast in `neg#1` and `neg#2`, and the calls to `neg` that assign to `a` and `b`, instead of the VCs (12), (13), (14) and (15), we get the respective Horn constraints:

$$(\kappa_1\,[\texttt{flag}/\nu] \wedge \mathit{true} \wedge \texttt{flag} = 0) \Rightarrow\ \nu = \texttt{x}\ \Rightarrow\ \mathit{false} \tag{17}$$
$$(\kappa_4\,[\texttt{flag}/\nu] \wedge \mathit{true} \wedge \texttt{flag} \neq 0) \Rightarrow\ \nu = \texttt{x}\ \Rightarrow\ \mathit{false} \tag{18}$$
$$\mathit{true} \Rightarrow\ \nu = 1\ \Rightarrow\ \kappa_1 \tag{19}$$
$$\mathit{true} \Rightarrow\ \nu = 0\ \Rightarrow\ \kappa_4 \tag{20}$$

These constraints are identical to the corresponding VCs except that κ variables appear in place of the unknown refinements for the corresponding binders. We can solve these constraints using fixpoint computations over a variety of abstract domains such as monomial predicate abstraction [13, 23] over a set of ground predicates which are arithmetic (in)equalities between program variables and constants, to obtain a solution mapping each κ to a concrete refinement:

$$\kappa_1 \doteq \nu = 0 \qquad \kappa_4 \doteq \nu \neq 0 \qquad \kappa_2, \kappa_3, \kappa_5, \kappa_6 \doteq \mathit{true}$$

which, when plugged back into the templates, allow us to infer types for `neg`.

Source Language: Syntax

$$
\begin{array}{rrcl}
\textit{Values} & v & ::= & \mathsf{c} \mid x \mid \lambda x.e \\
\textit{Expressions} & e & ::= & v \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid e\ \texttt{?}\ e_1\ \texttt{:}\ e_2 \mid e_1\ e_2 \\
\textit{Primitive Types} & \mathbb{B} & ::= & \mathsf{Num} \mid \mathsf{Bool} \\
\textit{Types} & A, B & ::= & \mathbb{B} \mid A \to B \mid A \wedge B \mid A \vee B
\end{array}
$$

Source Language: Operational Semantics $\boxed{e \longrightarrow e'}$

E-ECtx
$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

E-App-1
$$\mathsf{c}\ v \longrightarrow [\![\mathsf{c}]\!](v)$$

E-App-2
$$(\lambda x.e)\ v \longrightarrow [v/x]\ e$$

E-Cond-True
$$\mathtt{true}\ \texttt{?}\ e_1\ \texttt{:}\ e_2 \longrightarrow e_1$$

E-Cond-False
$$\mathtt{false}\ \texttt{?}\ e_1\ \texttt{:}\ e_2 \longrightarrow e_2$$

E-Let
$$\mathtt{let}\ x = v\ \mathtt{in}\ e \longrightarrow [v/x]\ e$$

Figure 5 Syntax and Operational Semantics of $\lambda^{\wedge}_{\vee}$.

Higher-Order Verification. Our two-phased approach generalizes directly to offer precise analysis for *polymorphic, higher-order* functions. Returning to the code in Figure 1, our two-phased inference algorithm infers the refinement types:

$$
\begin{aligned}
\mathtt{\$reduce} &:: \forall A, B.(a\!:\!A[\,], f\!:\!(B, A, \mathtt{idx}\langle a\rangle) \Rightarrow B, x\!:\!B) \Rightarrow B \\
\mathtt{reduce} &:: \forall A.(a\!:\!A[\,]^{+}, f\!:\!(A, A, \mathtt{idx}\langle a\rangle) \Rightarrow A) \Rightarrow A \\
&\quad \wedge \forall A, B.(a\!:\!A[\,], f\!:\!(B, A, \mathtt{idx}\langle a\rangle) \Rightarrow B, x\!:\!B) \Rightarrow B
\end{aligned}
$$

where $\mathtt{idx}\langle a\rangle$ describes *valid indices* for array a, and $A[\,]^{+}$ describes non-empty arrays:

$$
\begin{aligned}
\mathtt{idx}\langle a\rangle &\doteq \{\nu\!:\!\mathtt{number} \mid 0 \le \nu < \mathtt{len}(a)\} \\
A[\,]^{+} &\doteq \{\nu\!:\!A[\,] \mid 0 < \mathtt{len}(\nu)\}
\end{aligned}
$$

The above type is a precise *summary* for the higher-order behavior of `$reduce`: it describes the relationship between the input array a, the step ("callback") function f, and the initial value of the accumulator, and stipulates that the output satisfies the same *properties* B as the input x. Furthermore, it captures the fact that the callback f is only invoked on inputs that are valid indices for the array a that is being reduced. Consequently, Liquid Types [23], for example, would automatically infer:

$$
\begin{aligned}
\mathtt{step} &:: \forall A.(\mathtt{idx}\langle a\rangle, A, \mathtt{idx}\langle a\rangle) \Rightarrow \mathtt{idx}\langle a\rangle \\
\mathtt{minIndex} &:: \forall A.(A[\,]) \Rightarrow \mathtt{number}
\end{aligned}
$$

thereby verifying the safety of array accesses in the presence of higher order functions, collections, and value-based overloading.

3 Syntax and Operational Semantics of Rsc

Next, we formalize two-phase typing via a core calculus Rsc comprising a *source* language $\lambda^{\wedge}_{\vee}$ *with* overloading via union and intersection types, and a simply typed *target* language $\lambda^{\times}_{+}$ *without* overloading, where the assumptions for safe overloading are explicated via `DEAD`-casts. In Section 4, we describe the first phase that elaborates source programs into target programs,

Well-Formed Types $\boxed{\vdash A}$

$$\vdash \mathbb{B} \qquad \frac{\vdash A \quad \vdash B}{\vdash A \to B} \qquad \frac{\vdash A \quad \vdash B \quad \texttt{TAG}(A) = \texttt{TAG}(B)}{\vdash A \wedge B} \qquad \frac{\vdash A \quad \vdash B \quad \texttt{TAG}(A) \cap \texttt{TAG}(B) = \emptyset}{\vdash A \vee B}$$

$$\begin{array}{llll} \texttt{TAG}(\textsf{Num}) & = \{\texttt{"number"}\} & \texttt{TAG}(A \wedge A') & = \texttt{TAG}(A) \\ \texttt{TAG}(\textsf{Bool}) & = \{\texttt{"boolean"}\} & \texttt{TAG}(A \vee A') & = \texttt{TAG}(A) \cup \texttt{TAG}(A') \\ \texttt{TAG}(A \to A') & = \{\texttt{"function"}\} & & \end{array}$$

Figure 6 Basic Type Well-Formedness.

and finally, in Section 5 we describe how the second phase verifies the DEAD-casts on the target to establish the safety of the source. Our elaboration follows the overall compilation strategy of Dunfield [10] except that we have value-based overloading instead of an explicit "merge" operator [22], and consequently, our elaboration and proofs must account for source level "errors" via DEAD-casts.

3.1 Source Language ($\lambda^{\wedge}_{\vee}$)

Terms. We define a source language $\lambda^{\wedge}_{\vee}$, with syntax shown in Figure 5. Expressions include variables, functions, applications, let-bindings, a ternary conditional construct, and primitive constants c which include numbers $0, 1, \ldots$, operators $+, -, \ldots$, *etc.*

Operational Semantics. In figure 5 we also define a standard small-step operational semantics for $\lambda^{\wedge}_{\vee}$ with a left-to-right order of evaluation, based on evaluation contexts

$$E ::= \langle\, \rangle \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid E\ \texttt{?}\ e_1 : e_2 \mid E\ e \mid v\ E$$

Types. Figure 5 shows the types A in the source language. These include primitive types $\mathbb{B}$, arrow types $A \to B$ and, most notably, intersections $A \wedge B$ and (untagged) unions $A \vee B$ (hence the name $\lambda^{\wedge}_{\vee}$). Note that the source level types are *not* refined, as crucially, the first phase *ignores* the refinements when carrying out the elaboration.

Tags. As is common in dynamically typed languages, runtime values are associated with *type tags*, which can be inspected with a type test (cf. JavaScript's `typeof` operator). We model this notion to our static types, by associating each type with a set of possible tags. The multiplicity arises from unions. The meta-function $\texttt{TAG}(A)$, defined in Figure 6, returns the possible tags that values of type A may have at runtime.

Well-Formedness. In order to resolve overloads statically, we apply certain restrictions on the form of union and intersection types, shown by the judgment $\vdash A$ formalized in Figure 6. For convenience of exposition, the parts of an untagged union need to have distinct runtime tags, and intersection types require all conjuncts to have the same tag.

3.2 Target Language ($\lambda^{\times}_{+}$)

The target language $\lambda^{\times}_{+}$ eliminates (value-based) overloading and thereby provides a basic, well-typed skeleton that can be further refined with logical predicates. Towards this end,

Target Language: Syntax

$$
\begin{array}{llcl}
\textit{Expressions} & M, N & ::= & \texttt{c} \mid x \mid \lambda x.M \mid M \,?\, M_1 : M_2 \mid M_1\ M_2 \\
 & & \mid & (M_1, M_2) \mid \texttt{proj}_k M \mid \texttt{inj}_1\ M \mid \texttt{inj}_2\ M \\
 & & \mid & \texttt{case}\ M\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2 \mid \texttt{DEAD}_{A \downarrow B}\langle M \rangle \\
\textit{Values} & W & ::= & \texttt{c} \mid x \mid \lambda x.M \mid \texttt{inj}_1\ W \mid \texttt{inj}_2\ W \mid (M, M) \mid \texttt{DEAD}_{A \downarrow B}\langle W \rangle \\
\textit{Ref. Types} & T, S & ::= & \{\nu{:}\mathbb{B} \mid p\} \mid x{:}T \rightarrow S \mid T{+}S \mid T \times S
\end{array}
$$

Target Language: Operational Semantics $\boxed{M \longrightarrow M'}$

TE-ECtx
$$\frac{M \longrightarrow M'}{\mathcal{E}[M] \longrightarrow \mathcal{E}[M']}$$

TE-App-1
$$\frac{W \not\equiv \texttt{DEAD}_{A \downarrow B}\langle W' \rangle}{\texttt{c}\ W \longrightarrow [\![\texttt{c}]\!](W)}$$

TE-App-2
$$(\lambda x.M)\ W \longrightarrow [W/x]\ M$$

TE-Cond-True
$$\texttt{true} \,?\, M_1 : M_2 \longrightarrow M_1$$

TE-Cond-False
$$\texttt{false} \,?\, M_1 : M_2 \longrightarrow M_2$$

TE-Let
$$\texttt{let}\ x = W\ \texttt{in}\ M \longrightarrow [W/x]\ M$$

TE-Proj
$$\texttt{proj}_k(M_1, M_2) \longrightarrow M_k$$

TE-Case
$$\texttt{case}\ \texttt{inj}_k\ W\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2 \longrightarrow [W/x_k]\ M_k$$

Figure 7 Syntax and Operational Semantics of $\lambda_+^\times$.

unions and intersections are replaced with classical *tagged unions*, *products* and DEAD-casts, that encode the requirements for basic typing.

Terms. Figure 7 shows the terms M of $\lambda_+^\times$, which extend the source language with the introduction of pairs, projections, injections, a case-splitting construct and a special constant term $\texttt{DEAD}_{A \downarrow B}\langle M \rangle$ which denotes an erroneous computation. Intuitively, a $\texttt{DEAD}_{A \downarrow B}\langle M \rangle$ is produced in the elaboration phase whenever the actual type A for a term M is incompatible with an expected type B.

Operational Semantics. As in the source language we define evaluation contexts

$$
\begin{array}{rl}
\mathcal{E} ::= & \langle\,\rangle \mid \texttt{let}\ x = \mathcal{E}\ \texttt{in}\ M \mid \mathcal{E} \,?\, M_1 : M_2 \mid \mathcal{E}\ M \mid v\ \mathcal{E} \mid \texttt{inj}_k\ \mathcal{E} \\
\mid & \texttt{proj}_k \mathcal{E} \mid \texttt{DEAD}_{A \downarrow B}\langle \mathcal{E} \rangle \mid \texttt{case}\ \mathcal{E}\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2
\end{array}
$$

and use them to define a small-step operational semantics for the target in Figure 7. Note how evaluation is allowed in DEAD-casts and $\texttt{DEAD}_{A \downarrow B}\langle W \rangle$ *is* a value.

Types. The target language is checked against a refinement type checker. Thus, we modify the type language to account for the new language terms and refinements. *Basic Refinement Types* are of the form $\{\nu{:}\mathbb{B} \mid p\}$, consisting of the same basic types $\mathbb{B}$ as source types, and a logical predicate p (over some decidable logic), which describes the properties that values of the type must satisfy. Here, ν is a special *value variable* that describes the inhabitants of the type, that does not appear in the program, but can appear inside the refinement p. Function types are of the form $x{:}T \rightarrow S$, to express the fact that the refinement predicate of the return type S may refer to the value of the argument x. Sum and product types have the usual structure found in ML-like languages.

4 Phase 1: Trust

Terms of $\lambda^{\wedge}_{\vee}$ are elaborated to terms of $\lambda^{\times}_{+}$ by a judgment: $\Gamma \vdash e :: A \hookrightarrow M$. This is read: under the typing assumptions in Γ, term e of the source language is assigned a type A and elaborates to a term M of the target language. This judgment follows closely Dunfield's elaboration judgment [10], but with crucial differences that arise due to dynamic, value-based overloading, which we outline below.

Elaboration Ignores Refinements. A key aspect of the first phase is that elaboration is based solely on the basic types, *i.e.* does *not* take type refinements into account. Hence, the types assigned to source terms are transparent with respect to refinements; or more precisely, they work just as placeholders for refinements that can be provided as user specifications. These specifications are propagated *as is* during the first phase along with the respective basic types they are attached to. Due to this transparency of refinements we have decided to omit them entirely from our description of the elaboration phase.

4.1 Source Language Type-checking and Elaboration

Figure 8 shows the rules that formalize the elaboration process. At a high-level, following Dunfield [10], unions and intersections are translated to simpler typing constructs like sums and products (and the attendant injections, pattern-matches, and projections). Unlike the above work, which focuses on the classical intersection setting where overloading is explicit via a "merge" construct [22], we are concerned with the dynamic setting where overloading is value-based, leading to conventional type "errors".

Elaboration Modes: Strict and Flexible. Thus, one of the distinguishing features of our type system is its ability to not fail in cases where conventional static type system would raise type incompatibility errors, but instead elaborate the offending terms to the special error form $\texttt{DEAD}_{A|B}\langle M\rangle$. However, these error forms do not appear indiscriminately, but under certain conditions, specified by two elaboration modes: (1) a *flexible* judgment ($\vdash_F$) for rules that may yield $\texttt{DEAD}_{A|B}\langle M\rangle$ terms, and (2) a *strict* judgment ($\vdash_S$) for those that don't. Most elaboration rules come in both flavors, depending on the surrounding rules in a typing derivation. We write α to parameterize over the two modes.

Intuitively, we use flexible mode when checking calls to non-overloaded functions (with a *single* conjunct) and strict mode when checking calls to overloaded ones. In the former case, a type incompatibility truly signals a (potential) run-time error, but in the latter case, incompatibility may indicate the wrong choice of overload. Consequently, the elaboration judgment also states whether the intersection rule has been used, or not, by annotating the hook-arrow with the label y or n, respectively. As with strictness, we parametrize over n and y with the variable θ, and use $\star$ to denote that the outcome is not important.

Top-level Elaboration. Our top-level judgment is agnostic of either of the aforementioned modes. Elaborating programs in an empty context ($\vdash$) is essentially elaborating in the flexible sense and assumes we are not in the context of intersection elimination (T-TopLevel). Furthermore, an elaboration that succeeds in strict mode also succeeds in flexible mode (T-Weaken), so all strict rules can be used as flexible ones.

Standard Rules. Rules T-Cst, T-Var are standard and preserve the structure of the source program. Rule T-If expects the condition e of a conditional expression e ? e_1 : e_2 to

Elaboration Typing $\boxed{\Gamma \vdash e :: A \hookrightarrow M}$

$$\text{T-TopLevel}\ \frac{\cdot \vdash_F e :: A \overset{n}{\hookrightarrow} M}{\cdot \vdash e :: A \hookrightarrow M} \qquad \text{T-Weaken}\ \frac{\Gamma \vdash_S e :: A \overset{\theta}{\hookrightarrow} M}{\Gamma \vdash_F e :: A \overset{\theta}{\hookrightarrow} M}$$

$$\text{T-Cst}\ \ \Gamma \vdash_\alpha \texttt{c} :: \texttt{ty_c} \overset{\theta}{\hookrightarrow} \texttt{c} \qquad \text{T-Let}\ \frac{\Gamma \vdash_\alpha e_1 :: A_1 \overset{\star}{\hookrightarrow} M_1 \qquad \Gamma, x{:}A_1 \vdash_\alpha e_2 :: A_2 \overset{\theta}{\hookrightarrow} M_2}{\Gamma \vdash_\alpha \texttt{let}\ x = e_1\ \texttt{in}\ e_2 :: A_2 \overset{\theta}{\hookrightarrow} \texttt{let}\ x = M_1\ \texttt{in}\ M_2}$$

$$\text{T-Var}\ \frac{x{:}A \in \Gamma}{\Gamma \vdash_\alpha x :: A \overset{\theta}{\hookrightarrow} x} \qquad \text{T-If}\ \frac{\Gamma \vdash_F e :: \mathsf{Bool} \overset{n}{\hookrightarrow} M \qquad \forall i \in \{1,2\}\ .\ \Gamma \vdash_\alpha e_i :: A \overset{\theta}{\hookrightarrow} M_i}{\Gamma \vdash_\alpha e\ ?\ e_1 : e_2 :: A \overset{\theta}{\hookrightarrow} M\ ?\ M_1 : M_2}$$

$$\text{T-}\wedge\text{I}\ \frac{\forall k \in \{1,2\}\ .\ \Gamma \vdash_\alpha v :: A_k \overset{\theta}{\hookrightarrow} M_k \qquad \vdash A_1 \wedge A_2}{\Gamma \vdash_\alpha v :: A_1 \wedge A_2 \overset{\theta}{\hookrightarrow} (M_1, M_2)} \qquad \text{T-}\wedge\text{E}\ \frac{\Gamma \vdash_\alpha e :: A_1 \wedge A_2 \overset{\star}{\hookrightarrow} M}{\Gamma \vdash_\alpha e :: A_k \overset{y}{\hookrightarrow} \texttt{proj}_k M}$$

$$\text{T-Lam}\ \frac{\vdash A \to B \qquad \Gamma, x{:}A \vdash_\alpha e :: B \overset{\star}{\hookrightarrow} M}{\Gamma \vdash_\alpha \lambda x.e :: A \to B \overset{n}{\hookrightarrow} \lambda x.M} \qquad \text{T-App}\ \frac{\Gamma \vdash_\alpha e_1 :: A \to B \overset{y/n}{\hookrightarrow} M_1 \quad \Gamma \vdash_{S/F} e_2 :: A \overset{\star}{\hookrightarrow} M_2}{\Gamma \vdash_\alpha e_1\ e_2 :: B \overset{n}{\hookrightarrow} M_1\ M_2}$$

$$\text{T-}\bot\ \frac{\Gamma \vdash_F e :: A \overset{\theta}{\hookrightarrow} M \qquad \texttt{TAG}(A) \cap \texttt{TAG}(B) = \emptyset}{\Gamma \vdash_F e :: B \overset{\theta}{\hookrightarrow} \texttt{DEAD}_{A|B}\langle M \rangle} \qquad \text{T-}\vee\text{I}\ \frac{\Gamma \vdash_F e :: A_k \overset{\theta}{\hookrightarrow} M \qquad \vdash A_1 \vee A_2}{\Gamma \vdash_F e :: A_1 \vee A_2 \overset{\theta}{\hookrightarrow} \texttt{inj}_k\ M}$$

$$\text{T-}\vee\text{E}\ \frac{\Gamma \vdash_\alpha e_0 :: A_1 \vee A_2 \overset{\theta}{\hookrightarrow} M_0 \qquad \begin{array}{c}\Gamma, x_1{:}A_1 \vdash_\alpha E[x_1] :: B \overset{\theta}{\hookrightarrow} M_1 \\ \Gamma, x_2{:}A_2 \vdash_\alpha E[x_2] :: B \overset{\theta}{\hookrightarrow} M_2\end{array}}{\Gamma \vdash_\alpha E[e_0] :: B \overset{\theta}{\hookrightarrow} \texttt{case}\ M_0\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2}$$

Figure 8 Elaboration Typing rules.

be of boolean type, and assigns the same type A to each branch of the conditional. Rule T-Let checks expressions of the form `let` $x = e_1$ `in` e_2. It assigns a type A_1 to expression e_1 and checks e_2 in an environment extended with the binding of A_1 for x.

Intersections. In rule T-$\wedge$I the choice of the type we assign to a value v causes different elaborated terms W_k, as different typing requirements cause the addition of `DEAD`-casts at different places. This rule is intended to be used primarily for abstractions, so it's limited to accept values as input. Rule T-$\wedge$E for eliminating intersections replaces a term e that is originally typed as an intersection with a projection of that part of the pair that has a matching type. By T-$\wedge$I values typed at an intersection get a pair form.

Unions. Rule T-$\vee$I for union introduction is standard. The union elimination rule, taken from Dunfield's elaboration scheme [10], states that an expression e_0 can be assigned a union type $A_1 \vee A_2$ when placed at the "hole" of an evaluation context E, so long as the evaluation context can be typed with the same type B, when the hole is replaced with a variable typed

as A_1 on the one hand and as A_2 on the other. While the rule is inherently non-deterministic, it suffices for a declarative description of the elaboration process; see Dunfield's subsequent work on untangling type-checking of intersections and unions [9] for an algorithmic variant via a let-normal conversion.

Abstraction and Application. Rule T-Lam assumes the arrow type $A \to B$ is given as annotation and is required to conform to the well-formedness constraints. At the crux of our type system is the rule T-App. Expression e_1 can be typed in flexible mode. Depending on whether intersection elimination was used for e_1 we toggle on the mode of checking e_2. To only allow sensible derivations, we disallow the use of the `DEAD`-cast insertion when choosing among the cases of an intersection type. Below, we justify this choice using an example. If on the other hand, the type for e_1 is assigned without choosing among the parts of an intersection, then expression e_2 can be typed in flexible mode, potentially producing `DEAD`-casts.

Trusting via `DEAD`-Casts. The cornerstone of the "trust" phase lies in the presence of the T-⊥ rule. As we mentioned earlier, this rule can only be used in flexible mode. The main idea here is to allow cases that are obviously wrong, as far as the simple first phase type system is concerned; but, at the same time, include a `DEAD`-cast annotation and defer sound type-checking for the second phase. The premises of this rule specify that a `DEAD`-cast annotation will only be used if the inferred and the expected type have different tags. One of the consequences of this decision is that it does not allow `DEAD`-casts induced by a mismatch between higher-order types, as the tags for both types would be the same (most likely `"function"`). Thus, such mismatches are ill-typed and rejected in the first phase. This limitation is due to the limited information that can be encoded using the tag mechanism. A more expressive tag mechanism could eliminate this restriction but we omit this for simplicity of exposition.

Semantics of `DEAD`-Casts. To prove that elaboration preserves source level behaviors, our design of `DEAD`-casts preserves the property that the target gets stuck *iff* the source gets stuck. That is, source level type "errors" *do not lead to early* failures (*e.g.* at function call boundaries). Instead, `DEAD`-casts correspond to *markers* for all source terms that can potentially cause execution to get stuck. Hence, the target execution itself gets stuck at the same places as the source – *i.e.* when applying to a non-function, branching on a non-boolean or primitive application over the wrong base value, except that in the target, the stuckness can only occur when the value in question carries a `DEAD` marker. Consider the source program $(\lambda x.x\ 1)\ 0$ which gets stuck *after* the top-level application, when applying 1 to 0. It could be elaborated to $(\lambda x.x\ 1)\ \mathtt{DEAD}_{A\Downarrow B}\langle 0\rangle$ (where A and B are respectively Num and $\mathsf{Num} \to \mathsf{Num}$) which also has a top-level application and gets stuck at the second, inner application.

Necessity of elaboration modes. If we allowed the argument of an *overloaded* call-site to be checked in *flexible* context, then for the application $f\ x$, where f has been assigned the type $f : \mathtt{I} \to \mathtt{I} \wedge \mathtt{B} \to \mathtt{B}$ and $x : \mathtt{B}$, the following derivation would be possible:

$$\text{T-App}\ \dfrac{\text{T-}\wedge\text{E}\ \dfrac{\vdots}{\ldots \vdash_F\ f :: \mathtt{I} \to \mathtt{I} \overset{y}{\hookrightarrow} \mathtt{proj}_1 f} \qquad \text{T-}\bot\ \dfrac{\ldots \vdash_F\ x :: \mathtt{B} \overset{n}{\hookrightarrow} x \qquad \mathtt{TAG}(\mathtt{B}) \cap \mathtt{TAG}(\mathtt{I}) = \emptyset}{\ldots \vdash_F\ x :: \mathtt{I} \overset{n}{\hookrightarrow} \mathtt{DEAD}_{\mathtt{B}\Downarrow\mathtt{I}}\langle x\rangle}}{f : \mathtt{I} \to \mathtt{I} \wedge \mathtt{B} \to \mathtt{B}, x : \mathtt{B} \vdash_F\ f\ x :: \mathtt{I} \overset{n}{\hookrightarrow} (\mathtt{proj}_1 f)\ (\mathtt{DEAD}_{\mathtt{B}\Downarrow\mathtt{I}}\langle x\rangle)}$$

But, clearly, the intended derivation here is:

$$\text{T-App}\ \dfrac{\text{T-}\wedge\text{E}\ \dfrac{\vdots}{\ldots \vdash_F f :: \mathtt{B} \to \mathtt{B} \overset{y}{\hookrightarrow} \mathtt{proj}_2 f} \qquad \ldots \vdash_S x :: \mathtt{B} \overset{n}{\hookrightarrow} x}{f : \mathtt{I} \to \mathtt{I} \wedge \mathtt{B} \to \mathtt{B}, x : \mathtt{B} \vdash_F f\ x :: \mathtt{B} \overset{n}{\hookrightarrow} (\mathtt{proj}_2 f)\ x}$$

Subtyping. This formulation has been kept simple with respect to subtyping. The only notion of subtyping appears in the T-∨I rule, where a type A_1 is widened to $A_1 \vee A_2$. We could have employed a more elaborate notion of subtyping, by introducing a subtyping relation ($\leq$) and a subsumption rule for our typing elaboration. The rules for this subtyping relation would include, among others, function subtyping:

$$\dfrac{A_1' \leq A_1 \qquad A_2 \leq A_2'}{A_1 \to A_2 \leq A_1' \to A_2'}$$

However, supporting subtyping in higher-order constructs would only be possible with the introduction of wrappers around functions to accommodate checks on the arguments and results of functions. So, assuming that a cast c represents a dynamic check the above rule would correspond to a cast producing relation ($\triangleright$):

$$\dfrac{A_1' \triangleright A_1 \rightsquigarrow c_1 \qquad A_2 \triangleright A_2' \rightsquigarrow c_2}{A_1 \to A_2 \triangleright A_1' \to A_2' \rightsquigarrow \lambda f.\lambda x.(c_2\ (f\ (c_1\ x)))}$$

This formulation would just complicate the translation without giving any more insight in the main idea of our technique, and hence we forgo it.

4.2 Source and Target Language Consistency

In this section, we present the theorems that precisely connect the semantics of source programs with their elaborated targets. The main challenges towards establishing those are that: (1) the source and target do not proceed in lock-step, a single step of the one may be matched by several steps of the other (for example evaluating a projection in the target language does not correspond to any step in the source language), and (2) we must design the semantics of the `DEAD`-casts in the target to ensure that `DEAD`-casts cause evaluation to get stuck iff some primitive operation in the source gets stuck. We address these, next, with a number of lemmas and state our assumptions.

Value Monotonicity. This lemma fills in the mismatch that emerges when (non-value) expressions in the source language elaborate to values in the target language. Informally, if a source expression e elaborates to a target value W, then e evaluates (after potentially multiple steps) to a value v that is related to the target value W with an elaboration relation under the same type. Furthermore, all expressions on the path to the target value v elaborate to the same value and get assigned the same type.

▶ **Lemma 1** (Value Monotonicity). *If $\Gamma \vdash e :: A \hookrightarrow W$, then there exists v ·*

(1) $e \longrightarrow^* v$

(2) $\Gamma \vdash v :: A \hookrightarrow W$

(3) $\forall i\ s.t.\ e \longrightarrow^* e_i\ .\ \Gamma \vdash e_i :: A \hookrightarrow W$

Proof. The first two parts are handled similarly to Dunfield's [10] Lemma 11. The last part is proved by induction on the length of the path $e \longrightarrow^* e_i$. Details of this proof can be found in the extended version of this paper [30]. ◀

The reverse of the above lemma also comes in handy. Namely, given a value v that elaborates to an expression M and gets assigned the type A, there exists a value in the target language W, such that v elaborates to W and get assigned the *same* type A.

▶ **Lemma 2** (Reverse Value Monotonicity). *If* $\Gamma \vdash v :: A \hookrightarrow M$*, then exists* $W \cdot M \longrightarrow^* W$ *and* $\Gamma \vdash v :: A \hookrightarrow W$*.*

Proof. Similar to proof of Lemma 1. ◀

This is an interesting result as it establishes that different derivations may assign the same type to a term and still elaborate it to different target terms. For example, one can assume derivations that consecutively apply the intersection introduction and elimination rules. It's easy to see that the same value v can be used in the following elaborations:

$$\cdot \vdash v :: A_1 \wedge A_2 \hookrightarrow (W_1, W_2)$$

$$\cdot \vdash v :: A_1 \wedge A_2 \hookrightarrow \underbrace{(\texttt{proj}_1(W_1, W_2), \texttt{proj}_2(W_1, W_2))}_{M}$$

Lemma 2 guarantees it will always be the case that $M \longrightarrow^* (W_1, W_2)$. It is up to the implementation of the type-checking algorithm to produce an efficient target term.

Primitive Semantics. To connect the failure of the `DEAD`-casts with source programs getting stuck, we assume that the primitive constants are well defined for all the values of their input domain *but not* for `DEAD`-cast values. This lets us establish that primitive operations c are invariant to elaboration. Hence, a source primitive application gets stuck iff the elaborated argument is a `DEAD`-cast. The forward version of this statement is the following assumption.

▶ **Assumption 1** (Primitive constant application). *If* (1) $\cdot \vdash \mathsf{c} :: A \rightarrow B \hookrightarrow \mathsf{c}$, (2) $\cdot \vdash v :: A \hookrightarrow W$*, and* (3) $W \not\equiv \mathit{DEAD}_{\downarrow A}\langle\cdot\rangle$*, then* (i) $\mathsf{c}\ v \longrightarrow [\![\mathsf{c}]\!](v)$, (ii) $\mathsf{c}\ W \longrightarrow [\![\mathsf{c}]\!](W)$*, and* (iii) $\cdot \vdash [\![\mathsf{c}]\!](v) :: B \hookrightarrow [\![\mathsf{c}]\!](W)$*.*

Substitution lemma. The proof of soundness relies upon the following substitution lemma.

▶ **Lemma 3** (Substitution). *If* $\Gamma, x{:}A \vdash e :: A' \hookrightarrow M$ *and* $\Gamma \vdash v :: A \hookrightarrow W$ *then* $\Gamma \vdash [v/x]\ e :: A' \hookrightarrow [W/x]\ M$*.*

Proof. Similar to Dunfield's substitution proof [10] (Lemma 12). ◀

We use the above lemmas and assumptions to obtain a consistency result, analagous to Dunfield's Consistency Theorem [10], which states that the elaboration produces terms that are *consistent* with the source in that each step of the target is matched by a corresponding step of the source, *i.e.* the behaviors of the target *under-approximate* the behaviors of the source.

▶ **Theorem 4** (Consistency). *If* $\cdot \vdash e :: A \hookrightarrow M$ *and* $M \longrightarrow M'$ *then there exists* e' *such that* $e \longrightarrow^* e'$ *and* $\cdot \vdash e' :: A \hookrightarrow M'$*.*

Proof. The proof of this theorem is by induction on the derivation $\cdot \vdash e :: A \hookrightarrow M$, adapting the proof scheme given by Dunfield [10], and using Lemma 1. Details of this proof can be found in the extended version of this paper [30]. ◀

While this suffices to prove *soundness* – intuitively if the target does not "go wrong" then the source cannot "go wrong" either – it is not wholly satisfactory as a trivial translation that converts every source program to an ill-typed target also satisfies the above requirement. So, unlike Dunfield [10], we also establish a completeness result stating that if the source term steps, then the elaborated program will also eventually step to a corresponding (by elaboration) term. Theorem 5 declares that behaviors of the elaborated target *over-approximate* those of the source, and hence, in conjunction with Theorem 4, ensure that the source "goes wrong" iff the target does.

▶ **Theorem 5** (Reverse Consistency). *If* $\cdot \vdash e :: A \hookrightarrow M$ *and* $e \longrightarrow e'$ *then there exists* M' *such that* $\cdot \vdash e' :: A \hookrightarrow M'$*, and* $M \longrightarrow^{+} M'$*.*

Proof. Similar to the proof of Theorem 4, using adapted versions of the lemmas used by Dunfield [10] and Lemma 2. Again, details can be found in the accompanying report [30]. ◀

5 Phase 2: Verify

At the end of the first phase, we have elaborated the source with value based overloading into a classically well-typed target with conventional typing features and `DEAD`-casts which are really assertions that explicate the *trust assumptions* made to type the source. Thanks to Theorems 4 and 5 we know the semantics of the target are equivalent to the source. Thus, to verify the source, all that remains is to prove that the target will not "go wrong", that is to prove that the `DEAD`-casts are indeed never executed at run-time.

One advantage of our elaboration scheme is that at this point *any* program analysis for ML-like languages (*i.e.* supporting products, sums, and first class functions) can be applied to discharge the `DEAD`-cast [8]: as long as the target is safe, the consistency theorems guarantee that the source is safe. In our case, we choose to instantiate the second phase with *refinement types* as they: (1) are especially well suited to handle higher-order polymorphic functions, like `minIndex` from Figure 1, (2) can easily express other correctness requirements, *e.g.* array bounds safety, thereby allowing us to establish not just type safety but richer correctness properties, and, (3) are automatically inferred via the abstract interpretation framework of Liquid Typing [23]. Next, we recall how refinement typing works to show how `DEAD`-cast checking can be carried out, and then present the end-to-end soundness guarantees established by composing the two phases.

5.1 Refinement Type-checking

We present a brief overview of refinement typing as the target language falls under the scope of existing refinement type systems [18], which can, after accounting for `DEAD`-casts, be reused *as is* for the second phase. Similarly, we limit the presentation to *checking*; *inference* follows directly from Liquid Type inference [23]. Figure 9 summarizes the refinement system. The type-checking judgment is $G \vdash M :: T$, where *type environment* G is a sequence of bindings of variables x to refinement types T and *guard predicates*, which encode control flow information gathered by conditional checks. As is standard [18] each primitive constant `c` has a refined type `ty_c`, and a variable x with type T is typed as $\mathsf{sngl}(T, x)$ which is $\{\nu{:}\mathbb{B} \mid \nu = x\}$ if T is a basic type $\mathbb{B}$ and T otherwise.

Checking `DEAD`-casts. The refinement system verifies `DEAD`-casts by treating them as special function calls, *i.e.* discharging them via the application rule R-App. Formally, $\texttt{DEAD}_{A|B}\langle M \rangle$

Refined Typechecking $\boxed{G \vdash M :: T}$

$$\text{R-SUB}\ \frac{G \vdash M :: T_1 \qquad G \vdash T_1 \sqsubseteq T_2}{G \vdash M :: T_2} \qquad \text{R-CST}\ G \vdash \texttt{c} :: \texttt{ty_c}$$

$$\text{R-VAR}\ \frac{x{:}T \in G}{G \vdash x :: \mathsf{sngl}(T, x)} \qquad \text{R-LET}\ \frac{G \vdash M_1 :: T_1 \qquad G, x{:}T_1 \vdash M_2 :: T_2}{G \vdash \texttt{let}\ x = M_1\ \texttt{in}\ M_2 :: T_2}$$

$$\text{R-IF}\ \frac{G \vdash M :: \mathsf{Bool} \qquad G, M \vdash M_1 :: T \qquad G, \neg M \vdash M_2 :: T}{G \vdash M\ ?\ M_1\ :\ M_2 :: T} \qquad \text{R-LAM}\ \frac{G, x : T_x; G \vdash M :: T}{G \vdash \lambda x.M :: T_x \to T}$$

$$\text{R-APP}\ \frac{G \vdash M_1 :: T_x \to T \qquad G \vdash M_2 :: T_x}{G \vdash M_1\ M_2 :: [M_2/x]\ T} \qquad \text{R-PAIR}\ \frac{\forall k \in \{1,2\}\ .\ G \vdash M_k :: T_k}{G \vdash (M_1, M_2) :: T_1 \times T_2} \qquad \text{R-PROJ}\ \frac{G \vdash M :: T_1 \times T_2}{G \vdash \texttt{proj}_k M :: T_k}$$

$$\text{R-INJ}\ \frac{G \vdash M :: T_k}{G \vdash \texttt{inj}_k\ M :: T_1{+}T_2} \qquad \text{R-CASE}\ \frac{G \vdash M :: T_1{+}T_2 \qquad G, x_1 : T_1 \vdash M_1 :: T \qquad G, x_2 : T_2 \vdash M_2 :: T}{G \vdash \texttt{case}\ M\ \texttt{of}\ \texttt{inj}_1\ x_1 \Rightarrow M_1 \mid \texttt{inj}_2\ x_2 \Rightarrow M_2 :: T}$$

Refinement Subtyping $\boxed{G \vdash T_1 \sqsubseteq T_2}$

$$\sqsubseteq\text{-BASE}\ \frac{\mathsf{Valid}([\![G]\!] \wedge [\![p]\!] \Rightarrow [\![p']\!])}{G \vdash \{\nu{:}\mathbb{B} \mid p\} \sqsubseteq \{\nu{:}\mathbb{B} \mid p'\}} \qquad \sqsubseteq\text{-FUN}\ \frac{G \vdash T'_x \sqsubseteq T_x \qquad G, x{:}T'_x \vdash T \sqsubseteq T'}{G \vdash (x : T_x) \to T \sqsubseteq (x : T'_x) \to T'}$$

Figure 9 Refined Type-checking.

is treated as call to:

$$\texttt{DEAD}_{A\downarrow B} :: \mathsf{Bot}([A]) \to \mathsf{Bot}([B])$$

The notation $[\cdot]$ denotes the elaboration of $\lambda^{\wedge}_{\vee}$ types to $\lambda^{\times}_{+}$ types [10]:

$$[\mathbb{B}] \doteq \mathbb{B} \quad [A \wedge B] \doteq [A] \times [B] \quad [A \vee B] \doteq [A]{+}[B] \quad [A \to B] \doteq [A] \to [B]$$

The meta-function $\mathsf{Bot}(T) \doteq \mathsf{Tx}(T,\ \mathit{false})$ where:

$$\begin{array}{lll@{\qquad}lll}
\mathsf{Tx}(\mathbb{B},\ r) & \doteq & \{\nu{:}\mathbb{B} \mid r\} & \mathsf{Tx}(S{+}T,\ r) & \doteq & \mathsf{Tx}(S,\ r){+}\mathsf{Tx}(T,\ r) \\
\mathsf{Tx}(S \to T,\ r) & \doteq & \mathsf{Tx}(S,\ \neg r) \to \mathsf{Tx}(T,\ r) & \mathsf{Tx}(S \times T,\ r) & \doteq & \mathsf{Tx}(S,\ r) \times \mathsf{Tx}(T,\ r)
\end{array}$$

Returning to rule R-APP for `DEAD`-casts and inverting, expression M gets assigned a refinement type T. For simplicity we assume this is a base type $\mathbb{B}$. Due to R-SUB we get the subtyping constraint: $G \vdash \{\nu{:}\mathbb{B} \mid p\} \sqsubseteq \{\nu{:}\mathbb{B} \mid \mathit{false}\}$, which generates the VC: $\mathsf{Valid}([\![G]\!] \wedge [\![p]\!] \Rightarrow [\![\mathit{false}]\!])$. This holds if the environment combined with the refinement in the left-hand side is inconsistent, which means that the gathered flow conditions are infeasible, hence dead-code [18]. Thus, the refinements statically ensure that the specially marked `DEAD` values are *never created at run-time*. As only `DEAD` terms cause execution to get stuck, the refinement verification phase ensures that the source is indeed type safe.

Conditional Checking. R-IF and R-CASE check each branch of a conditional or case splitting statement, by enhancing the environment with a guard (M or $\neg M$) or the right

binding ($x:T_1$ or $x:T_2$), that encode the boolean test performed at the condition, or the structural check at the pattern matching, respectively. Crucially, this allows the use of "tests" inside the code to statically verify `DEAD`-casts and other correctness properties. The other rules are standard and are described in the refinement type literature.

Correspondence of Elaboration and Refinement Typing. The following result establishes the fact that the type A assigned to a source expression e by elaboration and the type T assigned by refinement type-checking to the elaborated expression M are connected with the relation: $\lceil A \rceil = \|T\|$, where $\|T\|$ is merely a (recursive) elimination of all refinements appearing in T. The notation $\lceil \Gamma \rceil = \|G\|$ means that for each binding $x:A \in \Gamma$ there exists $x:T \in G$, such that $\lceil A \rceil = \|T\|$, and vice versa.

▶ **Lemma 6** (Correspondence). *If $\Gamma \vdash e :: A \hookrightarrow M$, $G \vdash M :: T$ and $\lceil \Gamma \rceil = \|G\|$, then $\lceil A \rceil = \|T\|$.*

Proof. By induction on pairs of derivations: $\Gamma \vdash e :: A \hookrightarrow M$ and $G \vdash M :: T$. Details of this proof can be found in the extended version of this paper [30]. ◀

The target language satisfies a progress and preservation theorem [18]:

▶ **Theorem 7** (Refinement Type Safety). *If $\cdot \vdash M \ : \ T$ then either M is a value or there exists M' such that $M \longrightarrow M'$ and $\cdot \vdash M' \ : \ T$.*

Proof. Given by Vazou *et al.* [29] for a similar language. ◀

5.2 Two-Phase Type Safety

We say that a source term e is *well two-typed* if there exists a source type A, target term M and target (refinement) type T such that: (1) $\cdot \vdash e :: A \hookrightarrow M$, and, (2) $\cdot \vdash M :: T$. That is, e is well two-typed if it elaborates to a refinement typed target. The Consistency Theorems 4 and 5, along with the Safety Theorem 7, yield end-to-end soundness: well two-typed terms do not get stuck, and step to well two-typed terms.

▶ **Theorem 8** (Two-Phase Soundness). *If e is well two-typed then, either e is a value, or there exists e' such that:*
(1) **(Progress)** $e \longrightarrow e'$
(2) **(Preservation)** *e' is well two-typed.*

Proof. By induction on pairs of derivations: $\Gamma \vdash e :: A \hookrightarrow M$ and $G \vdash M :: T$. Details are to be found in the extended version of this paper [30]. ◀

6 Related Work

We focus on the highlights of prior work relevant to the key points of our technique: static types for dynamic languages, intersections and union types, and refinement types.

Types for Dynamic Functional Languages. *Soft typing* [5] incorporates static analysis to statically type dynamic languages: whenever a program cannot be proven safe statically, it is not rejected, but instead runtime checks are inserted. Henglein and Rehof [17] build up on this work by extending soft typing's monomorphic typing to polymorphic coercions and providing a translation of Scheme programs to ML. These works foreshadow the notion of *gradual typing* [24] that allows the programmer to control the boundary between static

and dynamic checking depending on the trade-off between the need for static guarantees and deployability. Returning to purely static enforcement, Tobin-Hochstadt *et al.* [27, 28] formalize the support for type tests as *occurrence typing* and extend it to an inter-procedural, higher-order setting by introducing propositional *latent predicates* that reflect the result of tests in Typed Racket function signatures.

Types for Dynamic Imperative Languages. Thiemann [26] and Anderson *et al.* [1] describe early attempts towards static type systems for JavaScript, and Furr *et al.* [15] present DRuby, a tool for type inference for Ruby scripts. However, these systems do not handle value-based overloading (like TypeScript, DRuby allows overloaded specifications for external functions). *Flow typing* [16] and TeJaS [19] account for tests using flow analysis, bringing occurrence typing to the imperative JavaScript setting, but, unlike our approach, they restrict themselves to a *fixed* set of type-testing idioms (*e.g.* `typeof`), precluding *general* value-based overloading *e.g.* as in `reduce` from Figure 1.

Logics for Dynamic Languages. The intuition of expressing subtyping relations as logical implication constraints and using SMT solvers to discharge these constraints allows for a more extensive variety of typing idioms. Bierman *et al.* [3] investigate semantic subtyping in a first order language with refinements and type-test expressions. In *nested refinement types* [7], the typing relation itself is a predicate in the refinement logic and a feature-rich language of predicates accounts for heavily dynamic idioms, like run-time type tests, value-indexed dictionaries, polymorphism and higher order functions. While program logics allow the use of arbitrary tests to establish typing, the circular dependency between values and basic types leads to two significant problems in theory and practice. First, the circular dependency complicates the *metatheory* which makes it hard to add extra (basic) typing features (*e.g.* polymorphism, classes) to the language. Second, the circular dependency complicates the *inference* of types and refinements, leading to significant annotation overheads which make the system difficult to use in practice. In contrast, two-phase typing allows arbitrary type tests while enabling the trivial composition of soundness proofs and inference algorithms.

Intersection and Union Types. Central to our elaboration phase are intersection and union types: Pierce [21] indicates the connection between unions and intersections with sums and products, that is the basis of Dunfield's elaboration scheme [10] on which we build. However, Dunfield studies *static* source languages that use *explicit* overloading via a merge operator [22]. In contrast, we target *dynamic* source languages with implicit value based overloading, and hence must account for "ill-typed" terms via `DEAD`-casts discharged via the second phase refinement check. Castagna *et al.* [6] describe a $\lambda\&$-calculus, where functions are overloaded by combining several different branches of code. The branch to be executed is determined at run-time by using the arguments' typing information. This technique resembles the code duplication that happens in our approach, but overload resolution (*i.e.* deciding which branch is executed) is determined at runtime whereas we do so statically.

Refinement Types. DML [31] is an early refinement type system composing ML's types with a decidable constraint system. *Hybrid type checking* [18] uses arbitrary refinements over basic types. A static type system verifies basic specifications and more complex ones are defered to dynamically checked contracts, since the specification logic is statically undecidable. In these cases, the source language is well typed (ignoring refinements), and lacks intersections and unions. Our second phase can use Liquid Types [23] to infer refinements using predicate abstraction.

7 Conclusions and Future Work

In this paper, we introduce two-phased typing, a novel framework for analyzing dynamic languages where value-based overloading is ubiquitous. The advantage of our approach over previous methods is that, unlike purely type-based approaches [28], we are not limited to a fixed set of tag- or type- tests, and unlike purely program logic-based approach [7], we can decouple reasoning about basic typing from values, thereby enabling inference.

Hence, we believe two-phased typing provides an ideal foundation for building expressive and automatic analyses for imperative scripting languages like JavaScript. However, this is just the first step; much remains to achieve this goal. In particular we must account for the imperative features of the language. We believe that decoupling makes it possible to address this problem by applying various methods for tracking mutation and aliasing [32] in the *first phase*, and we intend to investigate this route in future work to obtain a practical verifier for TypeScript.

Acknowledgements. We would like to thank our anonymous reviewers for their feedback. This work was supported by NSF Grants CNS-1223850, CNS-0964702 and gifts from Microsoft Research.

References

1 Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for Javascript. In *Proc. of the 19th European Conf. on Object-Oriented Programming*, 2005.

2 Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.*, 33(2), February 2011.

3 Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic Subtyping with an SMT Solver. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, 2010.

4 Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3), June 2005.

5 Robert Cartwright and Mike Fagan. Soft Typing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1991.

6 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A Calculus for Overloaded Functions with Subtyping. In *Proc. of the 1992 ACM Conf. on LISP and Functional Programming*, 1992.

7 Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested Refinements: A Logic for Duck Typing. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

8 Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*, 1977.

9 Joshua Dunfield. Untangling Typechecking of Intersections and Unions. In *Proc. of the 5th Workshop on Intersection Types and Related Systems, ITRS 2010, Edinburgh, U.K., 9th July 2010*, 2010.

10 Joshua Dunfield. Elaborating Intersection and Union Types. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, 2012.

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

12 Asger Feldthaus and Anders Møller. Checking Correctness of TypeScript Interfaces for JavaScript Libraries. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Language and Applications*, 2014.
13 Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation Inference for Modular Checkers. *Inf. Process. Lett.*, 77(2-4), February 2001.
14 Robert W. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19, 1967.
15 Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static Type Inference for Ruby. In *Proc. of the 2009 ACM Symp. on Applied Computing*, 2009.
16 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing Local Control and State Using Flow Analysis. In *Proc. of the 20th European Conf. on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, 2011.
17 Fritz Henglein and Jakob Rehof. Safe Polymorphic Type Inference for a Dynamically Typed Language: Translating Scheme to ML. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, 1995.
18 Kenneth Knowles and Cormac Flanagan. Hybrid Type Checking. *ACM Trans. Program. Lang. Syst.*, 32(2), February 2010.
19 Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. TeJaS: Retrofitting Type Systems for JavaScript. In *Proc. of the 9th Symp. on Dynamic Languages*, 2013.
20 Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Soft Contract Verification. In *Proc. of the 19th ACM SIGPLAN International Conf. on Functional Programming*, 2014.
21 Benjamin C. Pierce. *Programming With Intersection Types, Union Types, and Polymorphism*. PhD thesis, Carnegie Mellon University, 1991.
22 John C. Reynolds. ALGOL-like Languages, Volume 1. In Peter W. O'Hearn and Robert D. Tennent, editors, *Algol-like Languages*, chapter Design of the Programming Language FORSYTHE. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.
23 Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid Types. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2008.
24 Jeremy Siek and Walid Taha. Gradual Typing for Objects. In *Proce. of the 21st European Conf. on Object-Oriented Programming*, 2007.
25 The Satisfiability Modulo Theories Library. `http://smt-lib.org`.
26 Peter Thiemann. Towards a Type System for Analyzing Javascript Programs. In *Proc. of the 14th European Conf. on Programming Languages and Systems*, 2005.
27 Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. of the 35th Annual ACM SIGPLAN-SIGACT Symp.on Principles of Programming Languages*, 2008.
28 Sam Tobin-Hochstadt and Matthias Felleisen. Logical Types for Untyped Languages. In *Proc. of the 15th ACM SIGPLAN International Conf. on Functional Programming*, 2010.
29 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement Types for Haskell. In *Proc. of the 19th ACM SIGPLAN International Conf. on Functional Programming*, 2014.
30 Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Trust but Verify: Two-Phase Typing for Dynamic Languages – Extended. `http://arxiv.org/abs/1504.08039`.
31 Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Proc. of the 26th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, 1999.
32 Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kiezun, and Michael D. Ernst. Object and Reference Immutability Using Java Generics. In *Proc. of the the 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on The Foundations of Software Engineering*, 2007.

Concrete Types for TypeScript

Gregor Richards[1], Francesco Zappa Nardelli[2], and Jan Vitek[3]

1 **University of Waterloo, Canada**
2 **Inria, France**
3 **Northeastern University, USA**

Abstract

TypeScript extends JavaScript with optional type annotations that are, by design, unsound and, that the TypeScript compiler discards as it emits code. This design point preserves programming idioms developers are familiar with, and allows them to leave their legacy code unchanged, while offering a measure of static error checking in parts of the program that have type annotations. We present an alternative design for TypeScript, one where it is possible to support the same degree of dynamism, but where types can be strengthened to provide hard guarantees. We report on an implementation, called StrongScript, that improves runtime performance of typed programs when run on a modified version of the V8 JavaScript engine.

1998 ACM Subject Classification F.3.3 Studies of Program Constructs – Type structure

Keywords and phrases Gradual typing, dynamic languages

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.76

1 Introduction

Perhaps surprisingly, a number of modern computer programming languages have been designed with intentionally unsound type systems. Unsoundness arises for pragmatic reasons, for instance, Java has a covariant array subtype rule designed to allow for a single polymorphic `sort()` implementation. More recently, industrial extensions to dynamic languages, such as Hack, Dart and TypeScript, have featured *optional type systems* [5] geared to accommodate dynamic programming idioms and preserve the behavior of legacy code. Type annotations are second class citizens intended to provide machine-checked documentation, and only slightly reduce the testing burden. Unsoundness, here, means that a variable annotated with some type `T` may, at runtime, hold a value of a type that is not a subtype of `T` due to unchecked casts, covariant subtyping, and untyped code. Implementations deal with this by ignoring annotations, emitting code where all types are erased, and reverting to a fully dynamic implementation. For example, TypeScript translates classes to JavaScript code without casts or checks. Unsurprisingly, the generated code neither enjoys performance benefits nor strong safety guarantees.

A *gradual type system* [21, 18] presents a safer alternative, as values that cross between typed and untyped parts of a program are tracked and a mechanism for assigning blame eases the debugging effort by pinpointing the origin of any offending value. But the added safety comes with a runtime overhead, a price tag that, for object-oriented programs, can be steep. Also, gradual types affect the semantics of programs; adding type annotations that are more restrictive than strictly necessary can cause runtime errors in otherwise correct programs.

We argue that programmers should be given the means to express how much type checking they want to take place in any particular part of their program. Depending on their choice, they should either be able to rely on the fact that type annotations will not introduce errors

© Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek;
licensed under Creative Commons License CC-BY

in well-tested and widely deployed dynamic code, or, if they select more stringent checks, they should have guarantees of the absence of type errors and improved performance.

This paper illustrates this idea with the design of a new type system for the TypeScript language. TypeScript is an extension to JavaScript from Microsoft that introduces classes, structural subtyping, and type annotations on properties, arguments and return types. Syntactically, our extension, which we call StrongScript, is minimal, it consists of a single type constructor for concrete types (written `!`).[1] Semantically the changes are more subtle. Our type system allows developers to choose between writing untyped code (i.e., all variables are of type `any` as in JavaScript), optionally typed code that does not affect the semantics of dynamic programs (i.e., no new dynamic errors), and concretely typed code that provides the traditional correctness guarantees but affects the semantics of dynamic code (i.e., types are retained by the compiler and used to optimize the program, new dynamic errors may show up). More specifically, the goals that guided design of StrongScript are:

- All JavaScript programs must be valid StrongScript programs and common programming idioms should be typeable.
- Optional types guarantee that variables are used consistently with their declarations; concrete types are sound up to down casts.
- Type information should improve performance in the context of a highly-optimizing virtual machine.
- Support checked casts which are central to many object oriented idioms.

One of the more subtle departures between our proposal and TypeScript is a switch to nominal subtyping for classes. The reasons for this change are pragmatic: generating efficient property access code for structural subtyping is not a solved problem, whereas it is well understood for nominal subtyping. Moreover, with nominal subtyping, we can reuse the existing JavaScript subtype test. Interfaces retain their structural subtyping rules and are erased at compile-time like in TypeScript. This yields a type system where any class name `C` can be used as an optional type, written `C`, or as a concrete type, written `!C`. While the former have a TypeScript-like semantics, variables typed with concrete types are guaranteed to refer to an object of class `C`, a class that extends `C`, or `null`. We exploit concrete type annotations and nominal subtyping to provide fast property access and efficient checked casts. Unannotated variables default to type `any`, ensuring that JavaScript programs are valid StrongScript programs.

The contribution of this paper are twofold:

- *Design and implementation.* We implemented StrongScript as an extension to the TypeScript compiler. All the TypeScript programs we have tried run without changes on our implementation. To get a measure of performance benefits we extended Google's V8 to provide fast access to properties through concretely typed variables and floating point math with no runtime checks. Preliminary results on a small number of benchmarks show speed ups up to 22%. We also provide some evidence that the type system is not overly restrictive, as it validates all the benchmarks from [16].
- *Formalization.* We propose *trace preservation* as a key property for the evolution of programs from untyped to typed. Informally adding a type annotation to a program is trace-preserving if the program's behavior is unaffected. We prove a *trace preservation theorem* for optional types: if expression e is untyped, and e' only differs by the addition of optional types, then e and e' evaluate to the same value. We do this within a core

[1] Implementation available at `http://plg.uwaterloo.ca/~dynjs/strongscript/`.

```
class Vector {
 constructor(public x       ,
             public y       ,
             public z       ) {}
 times(k          )          { return new Vector(k*this.x,k*this.y,k*this.z); }
 dot  (v          )          { return this.x*v.x+this.y*v.y+this.z*v.z; }
 mag  (           )          { return Math.sqrt(this.x*this.x+this.y*this.y+this.z*this.z); }
}
```

L1 Untyped Vector library class

```
class Vector {
 constructor(public x:  number,
             public y:  number,
             public z:  number) {}
 times(k:  number)           { return new Vector(k*this.x,k*this.y,k*this.z); }
 dot  (v:  Vector)           { return this.x*v.x+this.y*v.y+this.z*v.z; }
 mag  (          )           { return Math.sqrt(this.x*this.x+this.y*this.y+this.z*this.z); }
}
```

L2 Optionally typed Vector library class

```
class Vector {
 constructor(public x: !number,
             public y: !number,
             public z: !number) {}
 times(k:  number): !Vector { return new Vector(k*this.x,k*this.y,k*this.z); }
 dot  (v:  Vector): !number { return this.x*v.x+this.y*v.y+this.z*v.z; }
 mag  (          ): !number { return +Math.sqrt(this.x*this.x+this.y*this.y+this.z*this.z); }
}
```

L3 Partly concrete Vector library class

```
class Vector {
 constructor(public x: !floatNumber,
             public y: !floatNumber,
             public z: !floatNumber) {}
 times(k: !number): !Vector { return new Vector(k*this.x,k*this.y,k*this.z); }
 dot  (v: !Vector): !number { return this.x*v.x+this.y*v.y+this.z*v.z; }
 mag  (          ): !number { return +Math.sqrt(this.x*this.x+this.y*this.y+this.z*this.z); }
}
```

L4 Concrete Vector library class

Figure 1 Gradual insertion of type annotations to a `Vector` class in StrongScript.

calculus, in the style of λ_{JS} [13], that captures the semantics of the two kinds of class types. A *safety* theorem states that terms can only get stuck when evaluating a cast or when accessing a property from a `any` or optionally typed variable. We also show that our design support program evolution by proving a *strengthening theorem*: when a fully optionally typed program is annotated with concrete types, the program will be trace preserving.

The design of StrongScript is based on our previous work on Thorn where optional types were called *like types* [2, 24]. As with the formalization of Bierman et al. [1], we restrict our extensions to the features of TypeScript 0.9.1 [15], the last version before the addition of generics. Our implementation is based on TypeScript 1.4, but newer features are unmodified beyond assuring that they remain safe with respect to concrete types.

2 Motivating Example

We illustrate gradual typing, and give a brief preview of StrongScript, with an example adapted from the `raytrace` benchmark of Section 7. The program includes a `Camera`, an extract of which is in Figure 2-C1. The camera is a client of the library class, `Vector`, shown in Figure 1-L1. For this example, assume the classes are developed and maintained independently.

As is often the case in dynamic languages, the library and client start out untyped (Figure 1-L1 and Figure 2-C1). This leaves the software open to modifications, something

C1 Untyped client class

```
class Camera {
 public fwd        ;
 constructor(public k         , v         ) {
  this.fwd = v.times(k);
 }
}
```

C2 Optionally typed client class

```
class Camera {
 public fwd:  Vector;
 constructor(public k:  number, v:  Vector) {
  this.fwd = v.times(k);
 }
}
```

C3 Concrete client class

```
class Camera {
 public fwd: !Vector;
 constructor(public k: !number, v: !Vector) {
   this.fwd = v.times(k);
 }
}
```

Figure 2 Gradual insertion of types in `Camera`.

that can be beneficial when frequent change is anticipated, but also means that all operations are dynamic. Dynamic operations can fail and, if the virtual machine is unable to optimize them, are more costly.

To communicate intent and provide machine-checked documentation, the library designer may annotate fields and arguments with optional types (Figure 1-L2). Fields `x`, `y` and `z` are expected to hold numeric values and are annotated with the generic `number` type. The argument to `dot()` is expected to be another `Vector`, this is also recorded with an optional type.

Optional types impose no constraints on clients. Thus, an untyped camera (Figure 2-C1) can be used with a typed vector. The benefit of ascribing an optional type to a variable is that within the variable's scope, the compiler can detect misuse. For example, `k.x` is erroneous because `k` has optional type `number` and numbers do not have an `x` property. Optional types can also be added to the camera (Figure 2-C2) with similar benefits.

As optional types lack guarantees, developers may strengthen the invariants of their code by adding concrete types (Figure 1-L3). When fields are typed as concrete numbers, written `x:!number`, programmers (and the compiler) can rely on the fact that a variable like `x` will only ever refer an instance of numeric type or null. Return types of methods can typically be made concrete without affecting clients. Making arguments concrete, on the other hand, may require some changes. The call to `Math.sqrt()` triggers a compilation error because the standard library is optionally typed, as all of its methods are easily replaced in JavaScript and thus can't necessarily be trusted. We can either add cast to `!number`, written `<!number>`, or use the common JavaScript idiom of adding a unary plus operator to ensure that the result is indeed a number.

The last typing step involves adding concrete types to arguments of the methods of the `Vector` class (Figure 1-L4). Doing so introduces a compile error in the client (Figure 2-C2) as argument `k` in the call `v.time(k)` is typed as an optional `number` whereas the method definition expects `!number`. The client must ensure that the argument is concrete (Figure 2-C3). One last point, to improve performance, StrongScript support hints like `!floatNumber` that give information about the expected storage format.

One important features of our design is that untyped classes can always interact with typed ones. The proper type checks is inserted by the implementation. Any combination

of library and client is valid. The only exception is that C2 will not compile with L4, this because C2 is typed but its types do not agree with L4.

3 Background and Related Work

The divide between static and dynamic types has fascinated academics and practitioners for years. Academics come determined to "cure the dynamic" as the absence of types is viewed as a flaw. Practitioners, on the other hand, seek to supplement their testing practices with machine-checked documentation and some ahead-of-time error checking. Dynamic languages are appreciated by practitioners for their support of exploratory programming, as any grammatically correct dynamic program, even a partial program or one with obvious errors, can be run, their productivity and their smaller learning curve. Decades of research were devoted to attempts to add static types to dynamic languages. In the 1980's, type inference and soft typing were proposed for Smalltalk and Scheme [20, 3, 7]. Inference based approaches turned out to be brittle as they required non-local analysis and were eventually abandoned.

Twenty years ago, while working at Animorphic on the virtual machine that would eventually become HotSpot, Bracha designed the first *optional type system* [6]. Subsequent work fleshed out the design [4] and detailed the philosophy behind optional types [5]. An optional type system is one that: (1) has no effect on the language's runtime semantics, and (2) does not mandate type annotations in the syntax. Strongtalk like Facebook's Hack, Google's Dart, and Microsoft's TypeScript was an industrial effort. In each case, a dynamic language is equipped with a static type system that is flexible enough to support backwards compatibility with untyped code. While optional types have benefits, they provide no guarantee of absence of type errors nor information that could be relied upon by an optimizing compiler.

Another important line of research is due to Felleisen and his collaborators. After investigating soft typing approaches for Scheme, Findler and Felleisen turned their attention to software contracts [9]. In [10], they proposed wrappers to enforce contracts for higher-order functions; these wrappers, higher-order functions themselves, were in charge of validating pre- and post-conditions and assigning blame in case of contract violations. Together with Flatt, they turned higher-order contracts into semantics casts [11]. A semantics cast consists of an argument (a value), a target type, and blame information. It evaluates to an object of the target type that delegates all behavior to its argument, and produces meaningful error messages in case the value fails to behave in a type appropriate manner. In 2006, Tobin-Hochstadt and Felleisen proposed a type system for, Typed Racket, a variant of Scheme that used higher-order contracts to enforce types at module boundaries [21]. Typed Racket has a robust implementation and is being used on large bodies of code [22]. The drawback of this approach is that contracts impose a runtime overhead which can be substantial in some programs.

In parallel with the development of Typed Racket, Siek and Taha defined *gradual typing* to refer to languages where type annotations can be added incrementally to untyped code [18, 19]. Like in Typed Racket, wrappers are used to enforce types but instead of focusing on module boundaries, any part of a program can be written in a mixture of typed and untyped code. The type system uses two relations, a subtyping relation and a consistency relation for assignment. Their work led to a flurry of research on issues such as bounding the space requirements for wrappers and how to precisely account for blame. In an imperative language their approach suffers from an obvious drawback: wrappers do not preserve object identity. One can thus observe the same object through a wrapper and through a direct reference at different types. Solutions are not appealing, either every property read must be checked or

	TypeScript	Typed Racket	Reticulated Python	StrongScript
x : C	any	W	W	any
x : !C	–	–	–	C
Trace preserving	●	○	○	◐
Fast property access	○	○	○	●

Figure 3 Optional and gradual type systems. This table's first line indicates possible values of variable declared of class `C`. This type is either `any` or `W` to denote the possibility of encountering a wrapper. The second line shows the possible value of variable declared `!C` in StrongScript, they are guaranteed to be unwrapped subtypes of that class. Trace preservation holds in TypeScript, in StrongScript developers can choose to forgo this property in exchange for stronger guarantees. The last line refers to the ability of a compiler to generate fast path code for property accesses.

fairly severe restrictions must be imposed on writes. In a Python implementation, called Reticulated Python, both solutions cause slowdowns that are larger than 2x [23]. Another drawback of gradual type systems is that they are not trace preserving. Consider:

```
class C:
  b = 41
def id( x:Object{b:String} ) -> Object{b:String}: return x
id( C() ).b + 1
```

Without annotations the program evaluates to `42`. When type annotations are taken into account it stops at the read of `b`. A type violation is reported as the required type for `b` is `String` while `b` holds an `Int`. Similar problems occur when developers put contracts that unnecessarily strong without understanding the range of types that can flow through a function.

The Thorn programming language was an attempt to combine optional types (called *like types*) with concrete types [2]. The type system was formalized along with a proof that wrappers can be compiled away [24]. Preliminary performance results suggested that concrete types could yield performance improvements when compared to a naive implementation of the language, but it was not demonstrated that the results hold for an optimizing compiler. Our work on StrongScript started as a straightforward port of the ideas to a different context, most of the differences are due to the details of TypeScript and JavaScript.

SafeTypeScript [16] is a recent effort from Microsoft to modify TypeScript by making it safe: in a nutshell, all types are concrete, and type checks are inserted when dynamic values are cast to concrete types. This technique yields a safe language which allows dynamic types, but lacks optional types. Because type checks are always inserted, SafeTypeScript is not trace-preserving and it lacks support for evolving programs from typed to untyped. On the other hand, SafeTypeScript focused on ensuring safety within the browser which is not a goal of our work.

Figure 3 compares the main approaches to adding types to dynamic languages. TypeScript chose to preserve the semantics of untyped code at the cost of guarantees and potential performance improvements. Typed Racket has an elegant type system that provides very strong guarantees of correctness. But the semantics of untyped code may be disrupted by too strong annotations and performance pathologies can cause serious slowdowns. Reticulated Python holds the promise of reducing the performance costs of gradual typing but does not deal with trace preservation. Lastly, StrongScript lacks some of the strong guarantees of Typed Racket (in particular about blame), but provides the means for programmers to write sound typed code and makes it easy for a compiler to generate code that is predictably efficient.

4 TypeScript: Unsound by design

Bierman et al. captured the key aspects of the design of TypeScript in [1]. TypeScript is a superset of JavaScript, with syntax for declaring classes, interfaces, and modules, and for optionally annotating variables, expressions and functions with types. Types are fully erased: errors not identified at compile-time will not be caught at runtime. The type system is structural rather than nominal, which causes some complications for subtyping. Type inference is performed to reduce the number of annotations. Some deliberate design decisions are the source of type holes, these include: unchecked casts, <String>obj is allowed if the type of obj is supertype of String, yet no check will be done at runtime; indexing with computed strings, obj[a+b] cannot be type checked as the value of string index is not known ahead of time; covariance of properties/arguments, this is similar to the Java array subtyping rule except that TypeScript does not have runtime checks for stores.

We will look more closely at the parts of the design that are relevant to our work, starting with subtyping. Consider the following well-typed program:

```
interface P { x: number; }
interface T { y: number; }
interface Pt extends P { y: number; dist(p: Pt); }
```

Interfaces include properties and methods. Extend declarations amongst interfaces are not required for other purposes than documenting intent, thus Pt is a subtype of both P and T. Classes can be defined as usual, and the extends clause there has a semantic meaning as it specifies inheritance of properties.

```
class Point {
  constructor (public x:number, public y:number){}
  dist(p: Point) { ... }
}
class CPoint extends Point {
  constructor (public color:String, x:number, y:number){
     super(x,y);
  }
  dist(p: CPoint) { ...p.color... }
}
```

Both classes are subtypes of the interfaces declared above. Note that the dist method is overridden covariantly at argument p and that CPoint.dist in fact does require p to be an instance of CPoint.

```
var o:Pt = new Point(0,0);
var c:CPoint = new CPoint("Red",1,1);
function pdist(x:Point, y:Point) { x.dist(y); }
pdist(c,o);
```

The first assignment implicitly casts Point to Pt which is allowed by structural subtyping. The function pdist will invoke dist at static types Point, yet it is invoked with a CPoint as first argument. The compiler allows the call, at runtime the access the p.color property will return the undefined value. Any type can be converted implicitly to any, and any method can be invoked on an any reference. More surprisingly, an any reference can be passed to all argument positions and be converted implicitly to any other type.

```
var q:any = new CPoint("Red",1,1);
var d = q.dist( o );
var b = o.dist( q );
```

This last example demonstrates a case of unchecked cast. Here `o` is declared of type `Pt` and we cast it to its subtype `CPoint`. The access will fail at runtime as variable `o` refers to an instance of `Point` which does not have color. The compiler does not emit a warning in any of the cases above.

```
function getc(x:CPoint) { return x.color };
getc( <CPoint>o );
```

Bierman et al. showed that the type system can be formalized as the combination of a sound calculus extended with unsound rules. For our purposes, the sound calculus is a system with records, equi-recursive types and structural subtyping. The resulting assignment compatibility relation can be defined coinductively using well-studied techniques [12]. We underline the critical choice of defining `any` as the supertype of all the types; since upcasts are well-typed, values of arbitrary types can be assigned to a variable of type `any` without the need of explicit casts. Type holes are introduced in three steps. First, a rule allows *downcasts* to subtypes. The second step is more interesting, as it changes the subtyping relation by stating that *all types are supertypes of* `any`. This implies arbitrary values can flow into typed variables without explicit casts. No syntactic construct identifies the boundaries between the dynamic and typed world. Thirdly, *covariant overloading* of class/interface members and methods is allowed.

Type inference is orthogonal to our proposal. As for generics, Bierman et al. describe decidability of subtyping as "challenging" [1]; we do not consider them here, and our implementation simply inserts runtime checks to assert their type safety. Lastly, we do not discuss TypeScript's liberal use of indexing. Our implementation supports it by explicitly inserting type casts (see Section 5).

5 StrongScript: Sound when needed

StrongScript builds on and extends TypeScript. Syntactically, the only addition is a new type constructor, written `!`. This yields three three kinds of type annotations:

Dynamic types, denoted by `any`, represent values manipulated with no static restrictions. Any object can be referenced by a `any` variable, all operations are allowed and any may fail at runtime.

Optional types, denoted by class names `C`, enable local type checking. All manipulations of optionally typed variables are verified statically against `C`'s interface. Optionally typed variables can reference arbitrary values, and so runtime checks are required to verify that those values conform to `C`'s interface.

Concrete types, denoted by `!C`, represent objects that are instance of the homonymous class or its subclasses. Static type checking is performed on these, and no dynamic checks are needed in the absence of downcasts.

Optional types have the same intent as TypeScript type annotations: they capture some type errors and enable features such as IDE name completion without reducing flexibility of dynamic programs. Concrete types behave exactly how programmers steeped in statically typed languages would expect. They restrict the values that can be bound to a variable and unlike other gradual type systems they do not support the notion of wrapped values or blame. No runtime error can arise from using a concretely typed variable and the compiler can rely on type informations to emit efficient code with optimizations such as unboxing and inlining.

To make good on the promise of concrete types, StrongScript has a sound type system. This forces some changes to TypeScript's overly permissive type rules and to the underlying implementation. The runtime thus distinguishes between *dynamic* objects, created with the JavaScript syntax `{ x:v .. }`, and objects which are instances of a class, created with the `new C(v..)` Java-like syntax. Casts are explicit and in many cases they require checks at runtime. Covariant subtyping, such as the array subtype rule, is checked at runtime as well. Moreover, class subtyping is nominal to ensure that the memory layout of parent classes is a prefix of child classes and thus that code to access properties is fast. Compared to TypeScript, subtyping is slightly simpler as we do not allow for `any` to be both the top and bottom of the type lattice. By design, any JavaScript program is a well-type StrongScript program, furthermore most TypeScript programs are also valid StrongScript programs – only in the rare cases discussed in Sec. 5.2 are TypeScript programs rejected by our type system (see also the evaluation in Sec. 7.2).

In what follows we introduce the main aspects of programming in our system. Code snippets should be read in sequence.

5.1 Programming with Concrete Types

We aim to let developers incrementally add types to their code, hardening parts that they feel need to be, while having the freedom to leave other parts dynamic. This is possible thanks to the interplay between the dynamic code, the flexible semantics of optionally typed variables, and the runtime guarantees of the concretely typed code. Consider the following program:

```
var p:any = { x=3; z=4 }
var f:any = func (p) {
  if (p.x < 10) return 10 else return p.distance() }
f(p)   // evaluates to 10
```

Without any loss of flexibility, programmers may choose to document their expectations about the argument of functions and data structures, and then annotate `p` and the argument of `f` with the optional type `Point`:

```
class Point {
  constructor(public x, public y){}
  dist(p) { return ... }
}
var p:Point = <any> { x=3; z=4 }  //Correct
var f:any = func (p:Point) {
  if (p.x < 10) return 0 else
  return p.distance(p) } //Wrong
```

Arbitrary objects can still flow into optionally typed variables, preserving flexibility (and ensuring trace-preservation), while the annotation of the argument of `f` enables local type checking, catching type errors such as the call to `distance`. The programmer can also create instances of class `Point`, which are concretely typed as `!Point`, and pass them to `f`:

```
var s:!Point = new Point(5,6);
f(s);  // evaluates to 10
```

As function `f` has been type checked assuming that its argument is a `Point`, we known it's body will manipulate the argument as a `Point`. However, whenever an object which is an instance of a class is passed to an optionally or dynamically typed context, it protects its

own abstractions at runtime. Consider a new class definition, where the `x` and `y` fields have been strengthened as `!number` and as such can only refer to instances of class `number`:

```
class TypedPoint {
  constructor(public x:!number, public y:!number){}
  dist(p) { return ... :!number }}

  var t:!TypedPoint = new TypedPoint(1,2);
  (<any>t).x = "o"  //DYNAMIC ERR: type mismatch
```

Some flexibility is lost by this class but the compiler can exploit the type informations to compute property offsets, remove runtime type checks and unbox values. Observe that dynamic, optional and concrete types can be mixed seamlessly; above for instance we have left the argument of the `dist` function dynamically typed, so that it is correct to invoke it with an arbitrary object as in `t.dist({x=1;y=2})`.

Our strategy for program evolution is to first add optional types, catching and fixing unexpected local type errors; the programmer can then identify the parts of the code that obey to a stricter type discipline, and replace optional types with concrete types. Optional types act as a bridge to move values into the concrete world:

```
  var fact = func(x:!number) {return ...:!number}
  var u:TypedPoint = { dist = function(p) {...} }
  var n:!number = fact(u.dist(p))
```

In the example, `p` has type `any`, and `u` points to a dynamic object with a method `dist` typed `any` → `any`. However, `u` has been typed as `TypedPoint`; the runtime will ensure that the method `dist` respects the `TypedPoint.dist` signature `any` → `!number` and will dynamically check that the returned value is an instance of class `number`. As a consequence, `fact(u.dist(p))` is well-typed (the concretely typed function `fact` is guaranteed to receive a value of type `!number`) and the programmer, by specifying just one optional type, can invoke the concretely-typed function `fact` with a value that has been computed from the dynamic world. The ability to have fine grained control over typing guarantees is one of the main benefits of StrongScript.

5.2 From TypeScript to StrongScript Types

A significant departure of our work is that we adopt nominal subtyping for classes and retain structural subtyping for interfaces and object literals. If a class `C` extends `D`, their concrete types are subtypes, denoted `!C` <: `!D`. Furthermore each concrete type is a subtype of the corresponding optional type, `!C` <: `C`, with an order on optional types that mirrors the one on concrete types: `!C` <: `!D` implies `C` <: `D`. `any` is an isolate with no super or subtype. Subtyping for interfaces follows [1] with the exception that an interface cannot extend a class.

Casts play a central role in the type system. Statically casts are always allowed to and from `any`, while casts to optional and concrete types are only permitted if one type is subtype of the other. At runtime, all programmer-inserted casts are checked, and additional casts are added by the implementation. Whenever a function with concretely typed arguments is injected in a dynamic context, the runtime adds a wrapper that uses casts to check the actual arguments. For instance, casting `fact` to `any` results in the following wrapper:

```
  func(x) { <any>(fact( <!number>x)) }
```

To keep the syntax of the two languages in sync, several TypeScript dynamic features are rewritten as implicit casts. In particular, at function arguments and the right hand side of the assignment operator, casts to or from `any` and optional types are inserted automatically. For instance, the expression on the left is transformed into the one on the right:

```
var p:Point = <any>{x=3, z=4}  var p:Point = <Point><any>{x=3, z=4}
```

If casts from `any` or optional types to concrete types are inserted, they are checked exactly like explicit casts. In addition, to support TypeScript's unsafe covariant subtyping, covariant overloading is implemented by injecting casts. Finally, casts are inserted in function calls to assure that if the function is called from an untyped context, its type annotations are honored. For instance, the class `CPoint` below extends `Point` and requires a concrete type for the argument of `dist`:

```
class CPoint extends Point {
  constructor(public color:string, x:number, y:number){...}
  dist(p:!CPoint) { ...p.color... }}
```

The overloading of `dist` is unsound, as `CPoint` is a subtype of `Point`. It is rewritten to perform a cast, and thus a check, on its argument `p`:

```
class CPoint extends Point {  ...
  dist(pa){var p:!CPoint = <!CPoint>pa; ...p.color...}}
```

Departing from TypeScript, the type of `this` is not `any`, but the concrete type of the surrounding class. This allows calls to methods of `this` to be statically type checked. But it creates an incompatibility with TypeScript code which uses "method stripping". It is possible to remove a method from the context of its object, and by using the builtin function `call`, to call the method with a different value for `this`. Consider, for instance, the following example:

```
class Animal {
  constructor(public nm:string) {}}
class Loud extends Animal {
  constructor(nm:string, public snd:string) { super(nm) }
  speak() { alert(this.nm+" says "+this.snd) }}

var a = new Animal("Snake");
var l = new Loud("Chris", "yo");
var m = l.speak;
m.call(a);
```

The `speak` method will be called with `this` referring to an `Animal`. This is plainly incorrect, but allowed, and will result in the string `"Chris says undefined"`. In StrongScript, `this` is concrete and the stripped method will include checks that cause the call to fail.

TypeScript's generic and union types are supported, but are not meaningfully checkable, and therefore may not be made concrete. Generics may reference concrete types and unions may include concrete types, however. For instance, the type `Array<!number>` is supported, but the type `!Array<number>` is not. Like other dynamic features, implicit casts are written to assure type safety at runtime.

TypeScript's enumeration types are treated as semantically identical to numbers.

5.3 Backwards compatibility

JavaScript allows a range of highly dynamic features. StrongScript does not prevent any of these features from being used, but, since their type behavior is so unpredictable, it does not

attempt to provide informative types for them. For instance, as objects are maps of string field names to values, it is possible to access members using computed strings. Thus `x[y]` accesses a member of `x` named by the string value of `y`, coercing it to a string if necessary; the type of the expression is always `any`. Assignment to `x[y]` may fail, if the member has a concrete type and the assigned value is not a subtype. Similarly, `eval` takes any string and executes it as code. StrongScript treats that code as JavaScript, not StrongScript. This is not an issue in practice as `eval`'s uses are mostly mundane [17]. The type of `eval(x)` is `any`.

Objects in JavaScript can be extended by adding new fields, and fields may be removed. An object's StrongScript type must be correct insofar as all fields and methods supported by its declared type must be present, but fields and methods *not* present in its type are unconstrained. As such, StrongScript protects its own fields from deletion or update to values of incorrect types, but does not prevent addition or deletion of new fields and methods. It is even possible to dynamically add new methods to classes, by updating an object prototype. None of this affects the soundness of the type system, and access to one of these in a value not typed `any` will result in a static type error.

5.4 Discussion

While one of previous our prototypes for StrongScript did implement blame tracking mode similar to Typed Racket, we decided to remove the feature as it did incur serious performance overheads. Wrappers require, amongst other things, specialized field access code. In Typed Racket the overheads are tolerable because the granularity of typing is coarser; wrappers are added when untyped values cross the boundary of a typed module. In our case, any method call is potentially a boundary. Fixing these performance issues is ongoing research. Our vision of blame tracking is as an optional command line switch like assertion checking to be used during debugging.

The change to nominal subtyping is controversial but practical experience suggests that structural subtyping is rather brittle.[2] In large systems, developed by different teams, the structural subtype relations are implicit and thus any small change in one part of the system could break the structural subtyping expected by another part of the system. We believe that having structural subtyping for optionally typed interface is an appropriate compromise. It should also be noted that Strongtalk started structural and switched to nominal [4].

StrongScript departs from Thorn inasmuch Thorn performs an optimized check on method invocation on optionally typed objects: rather than fully type checking the actual arguments against the method interface, it relied on the fact that this check had already been performed statically and simply compared the interface of the method invoked against the interface declared in the like type annotation. Thorn's type system is sound, but the simpler check introduces an asymmetry between optional and dynamic types at runtime which Thiemann exploited to prove that Thorn is not trace-preserving (personal communication).

[2] The TypeScript compiler (in `types.ts`) has the following comment: "*Note: 'brands' in our syntax nodes serve to give us a small amount of nominal typing. Consider 'Expression'. Without the brand, 'Expression' is actually no different (structurally) than 'Node'. Because of this you can pass any Node to a function that takes an Expression without any error. By using the 'brands' we ensure that the type checker actually thinks you have something of the right type. Note: the brands are never actually given values. At runtime they have zero cost.*" This suggests that the known drawbacks of structural subtyping do arise in practice.

6 Formal properties

We formalize StrongScript as an extension of the core language λ_{JS} of [13]; in particular we equip λ_{JS} with a nominal class-based type system à la Featherweight Java [14] and optional types. This treatment departs from Bierman et al. [1] in that they focused on typing interfaces and ignored classes, whereas we ignore interfaces and focus on classes. Thus our calculus does not include rules for structural subtyping of interface types; these rules would, assumedly, follow [1] but would add too much baggage to the formalization that is not directly relevant to our proposal. We also do not model method overloading (as discussed, StrongScript keeps covariant overloading sound by inserting appropriate casts) and references; our design enforces the runtime abstractions even in presence of aliasing.

Syntax. Class names are ranged over by C, D, the associated optional types are denoted by C and concrete types by !C, and the dynamic type is any. The function type $t_1 \mathinner{..} t_n \to t$ denotes explicitly typed functions while the type undefined is the type of the value `undefined`. The syntax of the language makes it easy to disambiguates class names from optional type annotations.

$$t \quad ::= \quad !\mathsf{C} \;\mid\; \mathsf{C} \;\mid\; \mathsf{any} \;\mid\; t_1 \mathinner{..} t_n \to t \;\mid\; \mathtt{undefined}$$

A program consists of a collection of class definitions plus an expression to be evaluated. A class definition $\mathtt{class}\,\mathsf{C}\,\mathtt{extends}\,\mathsf{D}\{s_1{:}t_1 \mathinner{..} s_k{:}t_k; md_1 \mathinner{..} md_n\}$ introduces a class named C with superclass D. The class has fields $f_1..f_k$ of types $t_1..t_k$ and methods $md_1..md_n$, where each method is defined by its name m, its signature, and the expression e it evaluates, denoted $m(x_1{:}t_1 \mathinner{..} x_k{:}t_k)\{\,\mathtt{return}\, e{:}t\}$. Type annotations appearing in fields and method definitions in a class definition cannot contain `undefined` or function types. Rather than baking base types into the calculus, we assume that there is a class String and a conversion function `toString`; string constants are ranged over by s. Expressions are inherited from λ_{JS} with some modifications (we often denote lists $l_1..l_n$ simply as $l_1..$):

$$\begin{array}{rcl} e & ::= & x \;\mid\; \{\; s_1{:}e_1 \mathinner{..} \mid t \;\} \;\mid\; e_{1\langle t\rangle}[e_2] \;\mid\; e_{1\langle t\rangle}[e_2] = e_3 \;\mid\; \mathtt{delete}\, e_1[e_2] \;\mid\; \langle t\rangle e \\ & \mid & \mathtt{new}\;\mathsf{C}\;(\,e_1..\,) \;\mid\; \mathtt{let}\;(x{:}t = e_1)\;e_2 \;\mid\; \mathtt{func}(x_1{:}t_1..)\{\mathtt{return}\;e : t\} \;\mid\; e(e_1..) \end{array}$$

Functions and let bindings are explicitly typed, expressions can be casted to arbitrary types, and the $\mathtt{new}\;\mathsf{C}\;(\,e_1..\,)$ expression creates a new instance of class C. More interestingly, objects, denoted $\{\; s_1{:}e_1 \mathinner{..} \mid t \;\}$, in addition to the fields' values, carry a type tag t: this is any for usual dynamic JavaScript objects, while for objects created by instantiating a class it is the name of the class. This tag enables preserving the class-based object abstraction at runtime. Additionally, field access (and, in turn, method invocation) and field update are annotated with the static type t of the callee e_1: this is used to choose the correct dispatcher or getter when executing method calls and field accesses, and to identify the cases where the property name must be converted into a string. These annotations can be added via a simple elaboration pass on the core language performed by the type checker.

Runtime abstractions. Two worlds coexist at runtime: fully dynamic objects, characterized by the any type tag, and instances of classes, characterized by the corresponding class name type tag. Dynamic objects can grow and shrink, with fields being added and removed at runtime, and additionally values of arbitrary types can be stored in any field, exactly as in JavaScript. The reduction rules confirm that on objects tagged any it is indeed possible to create and delete fields, and accessing or updating a field always succeeds.

$$\frac{}{\mathsf{!C} <: \mathsf{!Object}}\ [\textsc{SObject}] \qquad \frac{}{\mathsf{!C} <: \mathsf{C}}\ [\textsc{SOptInj}] \qquad \frac{\texttt{class C extends D } \{..\}}{\mathsf{!C} <: \mathsf{!D}}\ [\textsc{SClass}] \qquad \frac{t \neq \mathsf{!C}}{\mathsf{undefined} <: t}\ [\textsc{SUndef}] \qquad \frac{\mathsf{!C} <: \mathsf{!D}}{\mathsf{C} <: \mathsf{D}}\ [\textsc{SOptCov}]$$

$$\frac{t <: t' \quad t'_1 <: t_1\,..}{t_1 .. \to t <: t'_1 .. \to t'}\ [\textsc{SFunc}] \qquad \frac{\Gamma \vdash e : t_1 \quad t_1 <: t_2}{\Gamma \vdash e : t_2}\ [\textsc{TSub}] \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\ [\textsc{TVar}] \qquad \frac{}{\Gamma \vdash \texttt{undefined} : \mathsf{undefined}}\ [\textsc{TUndefined}]$$

$$\frac{\Gamma \vdash e : t_1 \quad t_1 = \mathsf{any} \vee t_2 = \mathsf{any} \vee t_1 <: t_2 \vee t_2 <: t_1}{\Gamma \vdash \langle t_2 \rangle e : t_2}\ [\textsc{TCast}] \qquad \frac{x_1{:}t_1 .., \Gamma \vdash e : t}{\Gamma \vdash \texttt{func}(x_1{:}t_1 ..)\{\texttt{return}\ e : t\} : t_1 .. \to t}\ [\textsc{TFunc}]$$

$$\frac{t = t' = \mathsf{any} \vee (t = \mathsf{C} \wedge t' = \mathsf{!C}) \quad t = \mathsf{C} \Rightarrow \Gamma \vdash e_1 : \mathsf{C}[s_1]\,..}{\Gamma \vdash \{\ s_1{:}e_1\,..\ |\ t\ \} : t'}\ [\textsc{TObj}] \qquad \frac{\mathsf{fields}(\mathsf{C}) = s_1{:}t_1\,.. \quad \Gamma \vdash e_1 : t_1\,..}{\Gamma \vdash \texttt{new}\ \mathsf{C}\ (\ e_1 ..\ ) : \mathsf{!C}}\ [\textsc{TNew}] \qquad \frac{\Gamma \vdash e_1 : t \quad x{:}t, \Gamma \vdash e_2 : t'}{\Gamma \vdash \texttt{let}\ (x{:}t = e_1)\ e_2 : t'}\ [\textsc{TLet}]$$

$$\frac{t = \mathsf{!C} \vee \mathsf{C} \quad \Gamma \vdash e : t}{\Gamma \vdash e_{\langle t \rangle}[s] : \mathsf{C}[s]}\ [\textsc{TGet}] \qquad \frac{\Gamma \vdash e_1 : \mathsf{any} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_{1\langle \mathsf{any} \rangle}[e_2] : \mathsf{any}}\ [\textsc{TGetAny}] \qquad \frac{\Gamma \vdash e : t_1 .. \to t \quad \Gamma \vdash e_1 : t_1\,..}{\Gamma \vdash e(e_1 ..) : t}\ [\textsc{TApp}] \qquad \frac{\Gamma \vdash e : \mathsf{any} \quad \Gamma \vdash e_1 : t_1\,..}{\Gamma \vdash e(e_1 ..) : \mathsf{any}}\ [\textsc{TAppAny}]$$

$$\frac{t = \mathsf{!C} \vee \mathsf{C} \quad \Gamma \vdash e_1 : t \quad \mathsf{not_function_type}(\mathsf{C}[s]) \quad \Gamma \vdash e_2 : \mathsf{C}[s]}{\Gamma \vdash e_{1\langle t \rangle}[s] = e_2 : t'}\ [\textsc{TUpdate}] \qquad \frac{\Gamma \vdash e_1 : \mathsf{any} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3}{\Gamma \vdash e_{1\langle \mathsf{any} \rangle}[e_2] = e_3 : \mathsf{any}}\ [\textsc{TUpdateAny}] \qquad \frac{\Gamma \vdash e_1 : \mathsf{any} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \texttt{delete}\ e_1[e_2] : \mathsf{any}}\ [\textsc{TDelete}]$$

$$\frac{\forall i.\ t_i \neq \texttt{undefined} \wedge t_i \neq t'_1 .. \to t' \quad \forall i.\ \vdash\ md_i \quad \{s_1 ..\} \cap \mathsf{fields}(\mathsf{D}) = \emptyset \ \wedge\ \{md_1 ..\} \cap \mathsf{methods}(\mathsf{D}) = \emptyset}{\vdash \texttt{class C extends D}\ \{\ s_1{:}t_1 ..;\ md_1 ..\ \}}\ [\textsc{TClass}] \qquad \frac{x_1{:}t_1 .. \vdash e : t}{\vdash m(x_1{:}t_1 ..)\{\texttt{return}\ e : t\}}\ [\textsc{TMethod}]$$

Figure 4 The type system.

In our design, objects which are instances of classes benefit from static typing guarantees; for instance, runtime type checking of arguments on method invocation is not needed as the type of the arguments has already been checked statically. For this, we protect the class abstraction: all fields and methods specified in the class interface must always be defined and point to values of the expected type. To understand how this is done, it is instructive to follow the life of a class-based object. The ENew rule implements the class pattern [8] commonly used to express inheritance in JavaScript. This creates an object with properly initialized fields – the type of the initialization values was checked statically by the TNew rule – and the methods stored in an object reachable via the `"__proto__"` field – the conformance of the method bodies with their interfaces is checked when typechecking classes, rules TClass and TMethod. For each method m defined in the interface, a corresponding function is

$$\frac{t = \mathsf{any} \vee \mathsf{C} \qquad \mathsf{tag}(v) \neq \mathsf{String}}{\{..\}_{\langle t \rangle}[v] \longrightarrow \{..\}_{\langle t \rangle}[\mathtt{toString}(v)]}\ [\textsc{EGetToString}]$$

$$\frac{t = \mathsf{any} \vee \mathsf{C} \qquad \mathsf{tag}(v) \neq \mathsf{String}}{\{..\}_{\langle t \rangle}[v] = v' \longrightarrow \{..\}_{\langle t \rangle}[\mathtt{toString}(v)] = v'}\ [\textsc{EUpdateToString}]$$

$$\frac{\mathsf{tag}(v_2) \neq \mathsf{String}}{\mathtt{delete}\ v_1[v_2] \longrightarrow \mathtt{delete}\ v_1[\mathtt{toString}(v_2)]}\ [\textsc{EDeleteToString}]$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}\ [\textsc{ECtx}]$$

$$\frac{s' \notin \{s..\} \qquad \mathtt{"__proto__"} \notin \{s..\}}{\{\ s{:}v\ ..\ |\ t\ \}_{\langle t' \rangle}[s'] \longrightarrow \mathtt{undefined}}\ [\textsc{EGetNotFound}]$$

$$\frac{s \notin \{s..\}}{\{\ \mathtt{"__proto__"}{:}v, s{:}v\ ..\ |\ t\ \}_{\langle t' \rangle}[s] \longrightarrow v_{\langle t' \rangle}[s]}\ [\textsc{EGetProto}]$$

$$\frac{s \in \mathsf{fields}(\mathsf{C})}{\{s{:}v\ ..\ |\ t\}_{\langle !\mathsf{C} \rangle}[s] \longrightarrow v}\ [\textsc{EGet}]$$

$$\frac{s \in \mathsf{fields}(\mathsf{C})}{\{s{:}v\ ..\ |\ t\}_{\langle \mathsf{C} \rangle}[s] \longrightarrow \langle \mathsf{C}[s] \rangle v}\ [\textsc{EGetOpt}]$$

$$\frac{}{\{s{:}v\ ..\ |\ t\}_{\langle \mathsf{any} \rangle}[s] \longrightarrow \langle \mathsf{any} \rangle v}\ [\textsc{EGetAny}]$$

$$\frac{\mathsf{tag}(v') <: \mathsf{C}[s] \vee s \notin \mathsf{fields}(\mathsf{C})}{\{s{:}v\ ..\ |\ \mathsf{C}\}_{\langle t \rangle}[s] = v' \longrightarrow \{s{:}v'\ ..\ |\ \mathsf{C}\}}\ [\textsc{EUpdate}]$$

$$\frac{}{\{s{:}v\ ..\ |\ \mathsf{any}\}_{\langle \mathsf{any} \rangle}[s] = v' \longrightarrow \{s{:}v'\ ..\ |\ \mathsf{any}\}}\ [\textsc{EUpdateAny}]$$

$$\frac{s_1 \notin \{s..\}}{\{\ s{:}v\ ..\ |\ t\ \}_{\langle t' \rangle}[s_1] = v' \longrightarrow \{\ s_1{:}v', s{:}v\ ..\ \ |\ t\ \}}\ [\textsc{ECreate}]$$

$$\frac{t = \mathsf{any} \vee (t = \ \mathsf{C} \wedge s \notin \mathsf{fields}(\mathsf{C}))}{\mathtt{delete}\ \{s{:}v\ ..\ |\ t\}[s] \longrightarrow \{..\ |\ t\}}\ [\textsc{EDelete}]$$

$$\frac{s \notin \{s_1..\} \vee (t = \ \mathsf{C} \wedge s \in \mathsf{fields}(\mathsf{C}))}{\mathtt{delete}\ \{\ s_1{:}v_1\ ..\ |\ t\ \}[s] \longrightarrow \{\ s_1{:}v_1\ ..\ |\ t\ \}}\ [\textsc{EDeleteNotFound}]$$

$$\frac{(t' = \mathsf{C} \wedge t' <: t) \vee (t = \mathsf{any} \vee \mathsf{C})}{\langle t \rangle \{..\ |\ t'\} \longrightarrow \{..\ |\ t'\}}\ [\textsc{ECastObj}]$$

$$\frac{t' = t'_1.. \to t'' \vee (t' = \mathsf{any} \wedge t'_1 = \mathsf{any}\ .. \wedge t'' = \mathsf{any})}{\begin{array}{l}\langle t' \rangle(\mathtt{func}(x_1{:}t_1..)\{\mathtt{return}\ e : t\}) \longrightarrow \\ \qquad \mathtt{func}(x_1{:}t'_1..)\{\mathtt{return}\ \langle t'' \rangle((\mathtt{func}(x_1{:}t_1..)\{\mathtt{return}\ e : t'\})(\langle t_1 \rangle x_1)..) : t''\}\end{array}}\ [\textsc{ECastFun}]$$

$$\frac{}{\mathtt{let}\ (x{:}t = v)\ e \longrightarrow e\{v/x\}}\ [\textsc{ELet}]$$

$$\frac{}{(\mathtt{func}(x_1{:}t_1..)\{\mathtt{return}\ e : t\})(v_1..) \longrightarrow e\{v_1/x_1..\}}\ [\textsc{EApp}]$$

$$\frac{}{\mathtt{new}\ \mathsf{C}\ (\ v_1..\ ) \longrightarrow \{\ \mathsf{gfields}\ \mathsf{C}\ (v_1..);\ \mathsf{gmethods}\ \mathsf{C}\ |\ \mathsf{C}\}}\ [\textsc{ENew}]$$

where, for $\mathtt{class}\ \mathsf{C}\ \mathtt{extends}\ \mathsf{D}\{s_1{:}t_1\ ..\ s_k{:}t_k;\ md_1\ ..\ md_n\}$, we define:

$$\begin{array}{l}\mathsf{gfields}\ \mathsf{C}\ (v_1..v_k\ v'..) \ \triangleq\ s_1{:}v_1..s_k{:}v_k;\ \mathit{fields}\ \mathsf{D}\ (v'..)\\ \mathsf{gmth}\ (m(x_1{:}t_1..)\{\mathtt{return}\ e : t\}) \ \triangleq\ \mathtt{"}m\mathtt{"} : \mathtt{func}(x_1{:}t_1..)\{\mathtt{return}\ e : t\}\\ \mathsf{gmethods}\ \mathsf{C} \triangleq \mathtt{"__proto__"} = \{\ \mathsf{gmth}\ md_1\ ..md_n;\ \mathsf{gmethods}\ \mathsf{D}\ |\ \mathsf{C}_{proto}\ \}\end{array}$$

Figure 5 The dynamic semantics.

stored in the prototype. The following type rules for method invocation can thus be derived from the rules for reading a field and applying a function:

$$\frac{t = !\mathsf{C} \vee \mathsf{C} \quad \Gamma \vdash e : t \quad \mathsf{C}[s] = t_1 \mathinner{..} t_n \to t' \quad \Gamma \vdash e_1 : t_1 \;\; .. \;\; \Gamma \vdash e_n : t_n}{\Gamma \vdash e_{\langle t \rangle}[s](e_1 \mathinner{..} e_n) : t'} \qquad \frac{\Gamma \vdash e : \mathsf{any} \quad \Gamma \vdash e' : t' \quad \Gamma \vdash e_1 : t_1 \;\; .. \;\; \Gamma \vdash e_n : t_n}{\Gamma \vdash e_{\langle \mathsf{any} \rangle}[e'](e_1 \mathinner{..} e_n) : \mathsf{any}}$$

The static view of the object controls the amount of type checking that must be performed at runtime. For this, field lookup $e_{\langle t \rangle}[e']$ is tagged at runtime with the static type t of e, as enforced by rules TGET and TGETANY. The absence of implicit subsumption to any guarantees that the tag is correct.

Suppose that the class Num implements integers and defines a method $+ : !\mathsf{Num} \to !\mathsf{Num}$. Let class C be:

$$\texttt{class}\ \mathsf{C}\{m(x{:}!\mathsf{Num})\{\,\texttt{return}\ x + 1{:}!\mathsf{Num}\}\}$$

Invoking m in a statically typed context directly passes the arguments to the method body:[3]

$$\begin{aligned}&(\texttt{new}\ \mathsf{C}(\,))_{\langle \mathsf{C} \rangle}[\texttt{"}m\texttt{"}](1) \xrightarrow{\text{ENew}} \{\texttt{"__proto__"}{:}\{\texttt{"}m\texttt{"}{:}v \mid !\mathsf{C}_{\mathsf{proto}}\} \mid !\mathsf{C}\}_{\langle !\mathsf{C} \rangle}[\texttt{"}m\texttt{"}](1)\\ &\xrightarrow{\text{EGetProto}} \{\texttt{"}m\texttt{"}{:}v \mid !\mathsf{C}_{\mathsf{proto}}\}_{\langle !\mathsf{C} \rangle}[\texttt{"}m\texttt{"}](1) \xrightarrow{\text{EGet}} v(1)\end{aligned}$$

where $v =$ $\texttt{func}\,(x{:}!\mathsf{Num})\{\,\texttt{return}\ x + 1{:}!\mathsf{Num}\}$. In a dynamic context, method invocation initially typechecks the arguments against the parameter type annotations of the method:

$$\begin{aligned}&(\langle \mathsf{any} \rangle \texttt{new}\ \mathsf{C}(\,))_{\langle \mathsf{any} \rangle}[\texttt{"}m\texttt{"}](1)\\ &\xrightarrow{\text{ENew}} (\langle \mathsf{any} \rangle \{\texttt{"__proto__"}{:}\{\texttt{"}m\texttt{"}{:}v \mid !\mathsf{C}_{\mathsf{proto}}\} \mid !\mathsf{C}\})_{\langle \mathsf{any} \rangle}[\texttt{"}m\texttt{"}](1)\\ &\xrightarrow{\text{ECast}} \{\texttt{"__proto__"}{:}\{\texttt{"}m\texttt{"}{:}v \mid !\mathsf{C}_{\mathsf{proto}}\} \mid !\mathsf{C}\}_{\langle \mathsf{any} \rangle}[\texttt{"}m\texttt{"}](1)\\ &\xrightarrow{\text{EGetProto}} \{\texttt{"}m\texttt{"}{:}v \mid !\mathsf{C}_{\mathsf{proto}}\}_{\langle \mathsf{any} \rangle}[\texttt{"}m\texttt{"}](1) \xrightarrow{\text{EGetAny}} (\langle \mathsf{any} \rangle v)(1)\\ &\xrightarrow{\text{ECastFun}} \langle \mathsf{any} \rangle (\texttt{func}\,(x{:}\mathsf{any})\{\,\texttt{return}\ v(\langle !\mathsf{Num} \rangle x){:}!\mathsf{Num}\}(1))\\ &\xrightarrow{\text{EApp}} \langle \mathsf{any} \rangle (v(\langle !\mathsf{Num} \rangle 1)) \xrightarrow{\text{ECast}} \langle \mathsf{any} \rangle (v(1))\end{aligned}$$

The expression above dynamically checks that the method argument argument is a !Num (last ECAST reduction) via the cast introduced by the combination of EGETANY and ECASTFUN rule. Observe that the choice of the rule EGETANY was guided by the tag any of the field access. The return value is injected back into the dynamic world via a cast to any, thus matching the corresponding static type rule. Contrast this with an invocation at the optional type D for some class D that defines a method m with type $!\mathsf{Num} \to t$:

$$\begin{aligned}&(\langle \mathsf{D} \rangle \texttt{new}\ \mathsf{C}(\,))_{\langle \mathsf{any} \rangle}[\texttt{"}m\texttt{"}](1) \xrightarrow{\text{ENew}}\xrightarrow{\text{ECast}}\xrightarrow{\text{EGetProto}} \{\texttt{"}m\texttt{"}{:}v \mid !\mathsf{C}_{\mathsf{proto}}\}_{\langle \mathsf{D} \rangle}[\texttt{"}m\texttt{"}](1)\\ &\xrightarrow{\text{EGetOpt}} (\langle !\mathsf{Num} \to t \rangle v)(1)\\ &\xrightarrow{\text{ECastFun}} \langle t \rangle (\texttt{func}\,(x{:}!\mathsf{Num})\{\,\texttt{return}\ v(\langle !\mathsf{Num} \rangle x){:}!\mathsf{Num}\}(1)) \xrightarrow{\cdots}\end{aligned}$$

[3] For simplicity we ignore the *this* argument. A preliminary λ_{JS}-like desugarer would rewrite the class definition as $\texttt{class}\ \mathsf{C}\{m(\mathit{this}{:}!\mathsf{C}, x{:}\mathsf{Num})\{\,\texttt{return}\ x+1{:}\mathsf{Num}\}\}$ and the method invocation as $\texttt{let}\ (o{:}!\mathsf{C} = \texttt{new}\ \mathsf{C}(\,))\ o_{\langle !\mathsf{C} \rangle}[\texttt{"}m\texttt{"}](o, 1)$.

In this case rule EGetOpt, selected via the D tag, inserts a cast to !Num $\rightarrow t$ that not only typechecks the actual arguments (as the caller can still an arbitrary object), but also casts the return value to the type t expected by the context.

Invariants of class-based objects are also enforced via the rule EDeleteNotFound, that turns deleting a field appearing in the interface of a class-based object into a no-op (which in static contexts is also forbidden by the TDelete rule), and rule EUpdate, that ensures that a field appearing in a class interface can only be updated if the type of the new value is compatible with the interface. For this, the auxiliary function $\mathsf{tag}(v)$ returns the type tag of an object, and is undefined on functions.

A quick inspection of the type rules shows that optionally-typed expressions – that is, expressions whose static type is C – are treated by the static semantics as objects of type !C, thus performing local type checking. At runtime, the reduction semantics highlights instead that optionally-typed objects are treated as dynamic objects except for return values. This ensures the third key property of optional types, namely that whenever field access or method invocation succeeds, the returned value is of the expected type and not any. We have seen how this is realized on method invocation; similarly for field accesses, let C be defined as $\texttt{class}\,\mathsf{C}\{"f":!\mathsf{Num}\}$ and compare the typing judgments $\{..\mid t\}_{\langle\mathsf{any}\rangle}["f"] : \mathsf{any}$ and $\{..\mid t\}_{\langle\mathsf{C}\rangle}["f"] : !\mathsf{Num}$. Field access on an object in a dynamic context invariably returns a value of type any. Instead if the object is accessed as C, then the rule TGet states that the type of the field access is !Num (which is then enforced at runtime by the cast inserted around the return value by rule EGetOpt).

Formalization. Once the runtime invariants are understood, the static and dynamic semantics is unsurprising. As usual, in the typing judgment for expressions, denoted $\Gamma \vdash e : t$, the environment Γ records the types of the free variables accessed by e. Object is a distinguished class name and is also the root of the class hierarchy; for each class name C we have a distinguished class name $\mathsf{C}_{\mathsf{proto}}$ used to tag the prototype of class-based objects at runtime. Function types are covariant on the return type, contravariant on the argument types: since the formalization does not support method overriding, it is sound for the *this* argument to be contravariant rather that invariant, which simplifies the presentation; the implementation supports overriding and imposes invariance of the *this* argument. Optional types are covariant and it is always safe to consider a variable of type !C as a variable of type C. The type rule for an object simply extracts its type tag, which, as discussed, is any for dynamic JavaScript objects,[4] and a class name for objects generated as instances of classes (possibly with the *proto* suffix). The notation $\mathsf{C}[s]$ returns the type of field s in class definition C; it is undefined if s does not belong to the interface of C. Auxiliary functions $\mathsf{fields}(C)$ and $\mathsf{methods}(C)$ return the set of all the fields and methods defined in class C (and superclasses). The condition $\mathsf{not_function_type}(\mathsf{C}[s])$ ensures that method updates in class-based objects are badly typed. Evaluation contexts are defined as follows:

$$\begin{array}{rcl} E & ::= & \bullet \mid \texttt{let}\,(x{:}t = E)e_2 \mid E_{\langle t\rangle}[e] \mid v_{\langle t\rangle}[E] \mid E[e_2] = e_3 \mid v[E] = e_3 \\ & \mid & v_1[v_2] = E \mid E(e_1 \mathinner{..} e_n) \mid v(v_1 \mathinner{..} v_n, E, e_1 \mathinner{..} e_k) \mid \texttt{new}\,\mathsf{C}(v_1 \mathinner{..} v_n\; E\; e_1\; e_k) \\ & \mid & \{s_1{:}v_1 \mathinner{..} s_n{:}v_n\; s{:}E\; s_1{:}e_1 \mathinner{..} s_k{:}e_k \mid t\} \mid \texttt{delete}\,E[e] \mid \texttt{delete}\,v[E] \mid \langle t\rangle E \end{array}$$

[4] Since the calculus does not formalize interfaces, it types dynamic object literals as any rather than with their implicit interface as done by TypeScript 1.4. The compiler described in Section 7 supports implicit interfaces.

As mentioned above, method invocation has higher priority than field access, and reduction under contexts (rule ECtx) should try to reduce $e_{\langle t \rangle}[e'](e_1)$ to $v_{\langle t \rangle}[v'](v_1)$ whenever possible.

Metatheory. In StrongScript, *values* are functions, and objects whose fields contain values. We say that an expression is *stuck* if it is not a value and no reduction rule applies; stuck expressions capture the state of computation just before a runtime error. The Safety theorem states that a well-typed expression can get stuck only on a downcast (as in Java) or on an optional-typed or dynamic expression.

▶ **Theorem 1** (Safety). *Given a well-typed program $\Gamma \vdash e : t$, if $e \longrightarrow^* e'$ and e' is stuck, then either $e' = E[\langle !C \rangle v'']$ and $\Gamma \vdash v'' : t''$ with $t'' \not<: !C$, or $e' = E[\{.. \mid t\}_{\langle t' \rangle}[v]]$ and $t' = \mathsf{any}$ or $t' = C$, or $e' = E[\langle t' \rightarrow t'' \rangle v'']$ and $\Gamma \vdash v'' : \mathsf{any}$ and v'' is not a function, or $e' = \mathtt{undefined}$.*

This theorem relies on two lemmas, the Preservation lemma states that typings (but not types) are preserved across reductions, and the Progress lemma identifies the cases above as the states in which well-typed terms can be stuck. The Safety theorem has several interesting consequences. First, a program in which all type annotations are concrete types has no runtime errors (apart from those occurring on downcasts): the concretely typed subset of StrongScript behaves as Featherweight Java (and, in turn, Java) and execution can be optimized along the same lines. Second, optional-typed programs (that is, programs with no occurrences of the any type and no downcasts to like types), benefit from the same execution guarantee: static type checking is strong enough to prevent runtime errors on entirely optional-typed programs.

The Trace Preservation theorem captures instead the idea that given a dynamic program, it is possible to add optional type annotations without breaking its runtime behavior; more precisely, if the type checker does not complain about the optional type annotation, then the runtime guarantees that the program will have the same behavior of the unannotated version. This theorem holds trivially in TypeScript because of type erasure.

▶ **Theorem 2** (Trace Preservation). *Let e be an expression where all type annotations are any and $\Gamma \vdash e : \mathsf{any}$. Let v be a value such that $e \longrightarrow^* v$. Let e' be e in which some type annotations have been replaced by optional type annotations (e.g. C, for C a class with no concrete types in its interface). If $\Gamma \vdash e' : t$ for some t, then $e' \longrightarrow^* v$.*

The Strengthening theorem states that if optional type annotations are used extensively, then the type checking performed is analogous to the type checking that would be performed by a strong type system à la Java. A consequence is that it is possible to transform a fully optionally typed program into a concretely typed program with the same behavior just by strengthening the type annotations. This property crucially relies on the fact that all source of unsoundness in our system are identified with explicit cast to optional types (or to any).

▶ **Theorem 3** (Strengthening). *Let e be a well-typed cast-free expression where all type annotations are of the form C or $!C$. Suppose that e reduces to the value v. Let e' be the expression obtained by rewriting all occurrences of optional types C into the corresponding concrete types $!C$. The expression e' is well-typed and reduces to the same value v.*

6.1 Assignability

The type system we formalized is picky about compatibility of types at binding. For instance, it rejects a program that passes a concretely typed object into a dynamic context as:

$\mathtt{let}\ (x{:}\mathsf{any} = \mathtt{new}\,\mathsf{Vector}(1,2,3))\ x_{\langle \mathsf{any} \rangle}[\texttt{"}\mathit{times}\texttt{"}](4)$

obtained by combining the concretely typed library of Figure 1-L4 with the untyped client of Figure 2-C1. Yet this program is correct. Our implementation tries to be more user-friendly. The StrongScript typechecker, part of the compiler described in the next section, inserts implicit type-casts, allowing some code to be accepted statically which would otherwise be rejected. We say that type t_1 is *assignable* to type t_2 if any of the following holds:

- t_1 is a subtype of t_2;
- t_1 or t_2 are any;
- t_1 is `number` and t_2 is `floatNumber` (and concrete cases thereof);
- t_1 is `number` or !`number` and t_2 is an enumeration type.

The last two cases are not relevant for the calculus but are supported by the implementation described in Section 7. The typechecker verifies the assignability relation on assignments, argument bindings, and return values (and, although not covered by the formalization, on comparisons (`==`, `<`, `>`, ...) and switch statements). If assignability does not hold, then the typechecker emits an error. Otherwise, in all the cases but the first, a cast to t_2 is inserted.

The above example implies an assignability check between !Vector and any, which holds because of the second case; a cast to any is inserted before the assignment:

$$\texttt{let}\ (x{:}\mathsf{any} = \langle\mathsf{any}\rangle\texttt{new}\,\mathsf{Vector}(1,2,3))\ x_{\langle\mathsf{any}\rangle}[\texttt{"}\mathit{times}\texttt{"}](4)$$

the resulting code is then checked following to the rules of Figure 4 (it is well-typed) and executed following Figure 5. Observe that assignability also enables interoperability between typed libraries and untyped clients. In the case below:

$$\texttt{let}\ (x{:}!\mathsf{Vector} = \langle\mathsf{any}\rangle\{..\})\ x_{\langle !\mathsf{Vector}\rangle}[\texttt{"}\mathit{times}\texttt{"}](4)$$

it is an implicit cast to !Vector which is inserted:

$$\texttt{let}\ (x{:}!\mathsf{Vector} = \langle !\mathsf{Vector}\rangle\langle\mathsf{any}\rangle\{..\})\ x_{\langle !\mathsf{Vector}\rangle}[\texttt{"}\mathit{times}\texttt{"}](4)$$

It is notable that this is the only one case of assignability implies a runtime type check: when t_1 is any and t_2 is a concrete type. Therefore assignability will not incur runtime checks in the absence of any-typed values. Assignability is also checked on type-casts, to emit warnings for “impossible” casts.

7 Evaluating StrongScript

Our implementation consists of two components: an extended version of the TypeScript 1.4 compiler and a JavaScript engine extended from Google’s V8 engine.[5][6] The compiler outputs portable JavaScript, so the resulting code can run on any stock virtual machine, but no performance improvement should be expected in that case. The compiler is extended with the following type related features: (a) support for concrete types and dynamic contracts at explicit downcasts, (b) checked downcasts where TypeScript does so implicitly and unsoundly (including covariant subtyping), and (c) function code suitable for both typed and untyped invocation (including dynamic contracts at untyped invocation). The compiler optionally

[5] https://developers.google.com/v8/

[6] The submitted version of this paper reported on a previous implementation in the Oracle Truffle VM [25]. The speed ups were similar, but raw performance was significantly below V8. Preparing the AEC artifact submission, we observed a memory leak that only manifested when running Truffle in a VMWare image. This prompted us to port our implementation to V8.

emits intrinsics that inform the runtime of monomorphic property accesses and known primitive types: we extended the V8 runtime to understand and exploit these intrinsics to perform check-free property access in concrete types and floating-point math with no runtime checks. It should be noted that our extensions to the V8 runtime are not exhaustive, they are meant to demonstrate the potential of type information. For instance, we do not attempt to unbox integer values, only floating point numbers. A richer implementation could get even better performance by generating optimized code for all TypeScript data types.

7.1 Implementation

Supporting concrete types simply requires adding the type constructor (`!`) and typing rules: `!C<:C` and `!C<:!D` implies `C<:D`. Since we use nominal typing for classes, optional and concrete types are compatible in both optional and concrete contexts; it is thus possible to implement type checks, using JavaScript's builtin `instanceof` mechanism. Nominal types are retained at runtime. The compiler ensures that concrete types are always used soundly. For this we include a small (200-line) library functions necessary to implement sound type checking. These functions rely on ECMAScript 5 features to protect themselves from being replaced or accidentally circumvented. To ensure soundness the compiler inserts dynamic contracts wherever unsafe downcasts occur, whether explicit or implicit. This is accomplished by the `$$check` function, which asserts that a value is of a specified type. For instance:

```
var untyped:any = new A();
var typed:!A = <!A>untyped;
```

is compiled into:

```
var untyped = new A();
var typed = A.$$check(untyped);
```

The check function is simple and generic, and does not require a per-class checker. For compatibility with TypeScript, several forms of unsafe, implicit casts are allowed in the source program. Specifically, implicit unsafe casts are inserted when a value is of type `any` and is in the context of a function argument or the right-hand-side of an assignment expression. For instance, the following code:

```
var unsafe:!B = <any>new A();
```

implies this additional cast:

```
var unsafe:!B = <!B><any>new A();
```

which in turn generates the following JavaScript code:

```
var unsafe = B.$$check(new A());
```

The cast to `!B` fails at runtime if `B` is not a supertype of `A`. Were this code to be rewritten with `unsafe:B` rather than `!B`, the inserted cast to `B` would imply no check, and the code would succeed at runtime. If the cast to `any` were omitted, this example would be rejected by the type checker.

Covariant overloading is implemented as unsafe downcasting, as described in Sec. 5.2. We describe some aspects of our type system as automatically-generated downcasts where TypeScript describes them as type compatibility. This is a matter of descriptive clarity and does not affect compatibility. Most semantically valid TypeScript 0.9.1 programs, and programs valid in TypeScript 1.0 and greater which use class types nominally and do not use features introduced after our version was forked from TypeScript, are semantically valid

StrongScript with no syntactic changes. Because some literals have concrete types (e.g. `0` has type `!number`), it is in some cases necessary to add explicit type annotations where implicit type annotations would choose inconsistent types (e.g. `number` in some cases and `!number` in others).

Efficient and sound implementation of function code. Functions with type annotations may be called from typed or untyped contexts. If they have only optional types or `any`, this requires no checks. However, methods of classes don not fit that description, as the `this` argument is always concretely typed. One option would be to check all concretely typed arguments at runtime, but this would entail unnecessary dynamic checks when types of arguments are known. Our implementation generates both an unchecked and a checked function. The checked function simply verifies its arguments and then calls the unchecked function. Calls are redirected appropriately by a compilation step. For instance, the following code:

```
class Animal {
  constructor(name:String) {}
  eat(x:!Animal) {
   console.log(this.name+" eats "+x.name); }}

var a:!Animal = new Animal("Alice");
var b:any = a;
a.eat(new Animal("Bob"));
b.eat(new Animal("Bob"));
```

is translated by an intermediary stage to:

```
class Animal {
  constructor(name:String) {}
  $$unchecked$$eat(x:!Animal) {
    console.log(this.name+" eats "+x.name); }
  eat(x) {
    (<!Animal>this).$$unchecked$$eat(<!Animal>x); }}

var a:!Animal = new Animal("Alice");
var b:any = a;
a.$$unchecked$$eat(new Animal("Bob"));
b.eat(new Animal("Bob"));
```

Code is generated to assure that the `$$unchecked` versions of functions are unenumerable and irreplaceable. This prevents accidental damage, but is not safe against intentionally malicious code.

Intrinsics. With concrete types, it is possible to lay out objects at compile time, and to access fields and methods by their statically-known location in the object layout, obviating the need for hash table lookups. JavaScript, however, provides no way to explicitly specify the layout of objects. Therefore, to take advantage of known concrete objects, JavaScript code generated by StrongScript may optionally include calls to several intrinsic operations which specify types and access fields by explicit offset within objects. On supporting implementations, these intrinsics are used to drive optimizations and eliminate guards. The intrinsics are `UnsafeAssumeMono`, `UnsafeAssumeFloat` and `ToFloat`, which support direct reading and writing to offsets within an object, check-free assertion of float values and forced coercion of numbers to IEEE floating-point values, respectively. Objects built using `UnsafeAssumeMono`

to write fields may be accessed by UnsafeAssumeMono, which modifies the behavior of the location cache: The location of the value is looked up by name in the first access, but following accesses cache the same location and do not perform runtime checks. Values created with ToFloat may be accessed with UnsafeAssumeFloat. This simply informs the JIT that the value will always be an IEEE floating-point value and does not require runtime type checks. Because JavaScript does not distinguish between floating-point numbers and integers, the type !floatNumber is supported, which is semantically identical to a number but hints the runtime that it should be stored as an IEEE float. For instance, the following StrongScript code:

```
class A { constructor(x:!floatNumber); }
var a:!A = new A(42);
alert(a.x * 3.14);
```

compiles into the following intrinsic-utilizing JavaScript code:

```
function A(x) { %_UnsafeAssumeMono(this.x = x); }
var a = new A(%_ToFloat(42));
alert(%_UnsafeAssumeFloat(%_UnsafeAssumeMono(a.x)));
```

7.2 Evaluating Performance

We measure the performance of our implementation to demonstrate that adding type information to dynamic code can yield performance benefits. For this experiment, we modified a small number of programs to give them concrete types and compared the result of running those on the V8 optimizing virtual machine against an untyped baseline. V8 is a highly optimizing, type-specializing compiler. Many of its optimizations are redundant with our intrinsics and we expect the relative speedups to reflect this fact.

As a baseline we used the benchmark suite provided by SafeTypeScript [16], which is in turn based on Octane[7]. We focused on programs which use classes, as our optimizations and type system rely on their presence. The benchmarks are crypto, navier-stokes, raytrace, richards and splay. These benchmarks were changed only by the addition of concrete types.

For each benchmark, a type erased and a typed form were compiled, called the "TypeScript" and "StrongScript" forms. Each benchmark times long-running iterative processes; several thousand iterations are performed before timing begins to allow the JIT a warmup period. We compare the runtime between the two forms on the same engine. i.e., the only change is the inclusion of intrinsics and type protection. Each benchmark involves a particular benchmark function looped 1000 times. We ran each of these benchmarks 10 times, interleaving executions of each benchmark and each form to reduce timing effects. We report the values in milliseconds each run. We report the arithmetic mean and standard deviations of the results in each form, as well as the speedup or slowdown. Benchmarks were run an 8-core 64-bit AMD FX(tm)-8320 with 16GB of RAM, running Ubuntu Linux. Our modification of V8 is based on a snapshot dated April 6, 2015. The SafeTypeScript benchmarks were compared against a snapshot of SafeTypeScript also dated April 6, 2015.

Results. Figure 6 shows that no benchmarks demonstrated slowdown outside of noise. Three of the benchmarks had speed up large enough to be statistically significant. The

[7] https://developers.google.com/octane/

Benchmark	Annot.	TypeScript runtime	TypeScript std dev	StrongScript runtime	StrongScript std dev	Speedup
crypto	76	19220	73	18089	45	6.25%
navier-stokes	116	15206	220	12609	204	20.59%
raytrace	73	48168	170	39380	144	22.32%
richards	35	38748	142	39082	142	-0.86%
splay	18	38273	302	37606	418	1.77%

Figure 6 Performance comparison on the V8 VM. Times are in milliseconds, lower is better.

performance benefits come from type-specialization intrinsics and direct access to fields in class instances. `crypto` uses primarily integer math, which does not presently benefit from our intrinsics, but also uses classes and displays a small but statistically significant speedup. `navier-stokes`[8] and `raytrace` use extensive floating point math as well as classes, and display benefits from both. Figure 6 also indicates the number of expressions, properties, and method arguments that had type annotations attached, which ranges from 18 to 116, and the standard deviations of both sets.

Threats to validity. The number of programs available and their nature makes it difficult to generalize form our results. At least they point to the potential for performance improvements with concrete types. Also, it is worthy of note that conventional wisdom amongst virtual machine designers is that type annotations are not needed to get performance for JavaScript. Our result suggest that this may not be the case. Because our intrinsics are unchecked JavaScript, it is possible to use them to circumvent security properties of the engine. Although this problem would be resolved by implementing StrongScript directly rather than through a translation layer, the performance characteristics of such a system may vary somewhat from what is achieved with a JavaScript system. Similar changes would be expected if StrongScript's specialized functions (e.g. `$$check` and `$$unchecked`) were made secure from malicious code. Our measured benchmark code has no unsafe downcasts, and thus no runtime type checking. The overall benefit of our intrinsics depends on the underlying engine, and specifically the precision of its speculation. Our intrinsics would be expected to show narrower advantages over an engine with better object layout speculation; however our intrinsics ensure *predictable* benefits, while layout speculation relies on complex heuristics that might be invalidated with program evolution.

8 Conclusion

StrongScript is a natural evolution of TypeScript. Optional type annotations have proven to be useful in practice despite their lack of runtime guarantees or performance benefits. With a modicum effort from the programmer, StrongScript can provide stronger runtime guarantees and predictable performance while allowing idiomatic JavaScript code and flexible program evolution. The type systems of TypeScript and StrongScript are fundamentally different, the former being intrinsically unsound for the stated goal of typing as many JavaScript programs

[8] `navier-stokes` displayed highly variable runtime in our initial configuration, with or without intrinsics, with time taken by a component of the V8 runtime beyond our control. We eliminated the warmup period for this benchmark to prevent this interference. Note that our intrinsics apply optimizations only to hot code, so this change does not benefit our runtime.

as possible, and the latter being sound to ensure stronger invariants when needed. In practice, we have found that StrongScript type system does not limit expressiveness as our compiler silently inserts all the needed casts to optional types or `any` needed to mimic the unsound behaviors of TypeScript. The only incompatibilities between the two are due to structural vs. nominal subtyping on optional class types. However all programs well-typed in versions of TypeScript up-to 0.9.1 – which relied on nominal subtyping – are well-typed StrongScript programs, and the large benchmarks of [16] suggest that this is not a problem in practice. Compared to SafeTypeScript, our design delivers the flexibility offered by the optional types and the predictable performance given by intrinsics. In particular, in our design, optional types are not only useful for program evolution but can also durably play the role of interfaces between the dynamic and concretely typed parts of a program, avoiding the need for extra casts to concrete types. The fact that we are able to achieve performance gains on a highly optimizing virtual machine gives one more reason for developers to adopt concrete types.

Acknowledgments. The authors are grateful to the following individuals and organizations: Adam Domurad for the V8 implementation, the ECOOP PC for accepting our paper, the ECOOP AEC for reviewing our artifact, Andreas Rossberg and Sam Tobin-Hochstadt for insightful comments, Nikhil Swamy for writing the SafeTypeScript benchmarks upon which our benchmarks are based, and the ANR (ANR-11-JS02-011), the NSF (SHF/1318227 and CSR/1523426) and the ONR for their financial support.

References

1. Gavin Bierman, Martin Abadi, and Mads Torgersen. Understanding TypeScript. In *ECOOP*, 2014. doi:10.1007/978-3-662-44202-9_11.
2. Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strnisa, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, concurrent, extensible scripting on the JVM. In *OOPSLA*, 2009. doi:10.1145/1639949.1640098.
3. Alan H. Borning and Daniel H. H. Ingalls. A type declaration and inference system for Smalltalk. In *POPL*, 1982. doi:10.1145/582153.582168.
4. Gilad Bracha. The Strongtalk type system for Smalltalk. In *OOPSLA Workshop on Extending the Smalltalk Language*, 1996.
5. Gilad Bracha. Pluggable type systems. *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.
6. Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*, 1993. doi:10.1145/165854.165893.
7. Robert Cartwright and Mike Fagan. Soft Typing. In *PLDI*, 1991. doi:10.1145/113446.113469.
8. Douglas Crockford. Classical inheritance in JavaScript. `http://www.crockford.com/javascript/inheritance.html`.
9. Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *OOPSLA*, 2001. doi:10.1145/504282.504283.
10. Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP*, 2002. doi:10.1145/581478.581484.
11. Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *ECOOP*, 2004. doi:10.1007/978-3-540-24851-4_17.
12. Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(6):511–548, 2002. doi:10.1017/S0956796802004318.

13 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *ECOOP*, 2010. doi:10.1007/978-3-642-14107-2_7.
14 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3), 2001. doi:10.1145/503502.503505.
15 Microsoft. TypeScript – language specification version 0.9.1. Technical report, Microsoft, August 2013.
16 Aseem Rastogi, Nikhil Swamy, Cedric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe and efficient gradual typing for TypeScript. In *POPL*, 2015. doi:10.1145/2676726.2676971.
17 Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *ECOOP*, 2011. doi:10.1007/978-3-642-22655-7_4.
18 Jeremy Siek. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.
19 Jeremy Siek and Walid Taha. Gradual typing for objects. In *ECOOP*, 2007. doi:10.1007/978-3-540-73589-2_2.
20 Norihisa Suzuki. Inferring types in Smalltalk. In *POPL*, 1981. doi:10.1145/567532.567553.
21 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *DLS*, 2006. doi:10.1145/1176617.1176755.
22 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed Scheme. In *POPL*, 2008. doi:10.1145/1328438.1328486.
23 Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for Python. In *DLS*, 2014. doi:10.1145/2661088.2661101.
24 Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *POPL*, 2010. doi:10.1145/1706299.1706343.
25 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 *Onwards!*, 2013. doi:10.1145/2509578.2509581.

Simple and Effective Type Check Removal through Lazy Basic Block Versioning

Maxime Chevalier-Boisvert and Marc Feeley

DIRO, Université de Montréal
Montréal, QC, Canada
`{chevalma,feeley}@iro.umontreal.ca`

Abstract

Dynamically typed programming languages such as JavaScript and Python defer type checking to run time. In order to maximize performance, dynamic language VM implementations must attempt to eliminate redundant dynamic type checks. However, type inference analyses are often costly and involve tradeoffs between compilation time and resulting precision. This has lead to the creation of increasingly complex multi-tiered VM architectures.

This paper introduces *lazy basic block versioning*, a simple JIT compilation technique which effectively removes redundant type checks from critical code paths. This novel approach lazily generates type-specialized versions of basic blocks on-the-fly while propagating context-dependent type information. This does not require the use of costly program analyses, is not restricted by the precision limitations of traditional type analyses and avoids the implementation complexity of speculative optimization techniques.

We have implemented intraprocedural lazy basic block versioning in a JavaScript JIT compiler. This approach is compared with a classical flow-based type analysis. Lazy basic block versioning performs as well or better on all benchmarks. On average, 71% of type tests are eliminated, yielding speedups of up to 50%. We also show that our implementation generates more efficient machine code than TraceMonkey, a tracing JIT compiler for JavaScript, on several benchmarks. The combination of implementation simplicity, low algorithmic complexity and good run time performance makes basic block versioning attractive for baseline JIT compilers.

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 Optimization, Type Checking, Code Generation, JavaScript

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.101

1 Introduction

A central feature of dynamic programming languages is that they defer type checking to run time. In order to maximize performance, efficient implementations of dynamic languages seek to type-specialize code so as to eliminate dynamic type checks when possible. Doing so requires proving that these type checks are unnecessary and generating type-specialized code.

Traditionally, the main approach for eliminating type checks has been to use type inference analyses. This is problematic for modern dynamic languages such as JavaScript and Python for three main reasons. The first is that these languages are generally poorly amenable to whole-program type analyses. Constructs such as `eval` and dynamic loading of modules can destroy previously valid type information. The second is that these analyses can be expensive in terms of computation time and memory usage, making them unsuitable for JIT compilers, particularly baseline compilers. To reduce analysis cost, it is often necessary to

© Maxime Chevalier-Boisvert and Marc Feeley;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 101–123

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

sacrifice precision. A last issue is that some type checks simply cannot be eliminated through analysis alone, without code transformations.

Because dynamic programming languages are generally poorly amenable to type inference, and whole-program analyses are often too expensive for JIT compilation purposes, state of the art JavaScript VMs such as SpiderMonkey, V8 and JavaScriptCore rely on increasingly complex multi-tiered architectures integrating interpreters and multiple JIT compilers with different optimization capabilities (baseline compilers to aggressively optimizing compilers). At the highest optimization levels, modern JIT compilers typically make use of type feedback, type inference analysis and also speculative optimization and deoptimization [16] with On-Stack Replacement (OSR).

We introduce a simple approach to JIT compilation that generates efficient type-specialized code without the use of costly type inference analyses or type profiling. The approach, which we call lazy basic block versioning, lazily clones and specializes basic blocks on-the-fly in a way that allows the compiler to accumulate type information while machine code is generated, without a separate type analysis pass. The accumulated information allows the removal of redundant type tests, particularly in performance-critical paths.

Lazy basic block versioning lets the execution of the program itself drive the generation of type-specialized code, and is able to avoid some of the precision limitations of traditional, conservative type analyses as well as avoiding the implementation complexity of speculative optimization techniques.

This paper relates our experience implementing lazy basic block versioning and reports on its effectiveness as a code generation technique. The rest of the paper is organized as follows. Section 2 explains the basic block versioning approach, comparing it with the related approaches of static type analysis and trace compilation. Section 3 describes an implementation within Higgs, an experimental JIT compiler for JavaScript. Section 4 presents an evaluation of the performance of this implementation. Related work is presented in Section 5.

2 Basic Block Versioning

In the basic block versioning approach, the code generator maintains a typing context (or type map) which indicates what is known of the type of each live local variable at the current program point. All local variables start out with the *unknown* type at function entry points. While generating code, the code generator updates the typing context by inferring the result type of data operations it encounters. Conditional branch instructions corresponding to type tests create two new typing contexts for outgoing branch edges. In each context, a type is assigned to the variable being tested (either the type tested or *unknown*). When a type test branch instruction is encountered and the type of the argument is known, the branch direction can be determined at code generation time and the type test eliminated.

The compiler may generate code for multiple instances of a given basic block; one version for each typing context encountered on a branch to that basic block. This allows specializing the basic block and its successors by taking the types of live variables into account. While basic block versioning works at the level of individual basic blocks, the propagation of typing contexts to successor blocks allows type-specializing entire control flow graphs.

An important difference between this approach and traditional type analyses is that basic block versioning does not compute a fixed point on types to be inferred. Variables which may have multiple different types at the same program point are handled more precisely with basic block versioning due to the duplication of code. In a traditional type analysis,

the union of several possible types would be assigned to such variables, causing the analysis to be conservative. With basic block versioning, distinct basic block versions, and thus distinct code paths, will be created for each type previously encountered, allowing a precise context-dependent tracking of types.

With basic block versioning, loops in the control flow graph need not be handled specially. A first version of the loop header is generated for a typing context C_1. At the point(s) where control flow branches back to the loop header, a new version of the loop may be generated if the typing context C_2 is different from C_1. Given that the number of possible contexts is finite, a fixed point is eventually reached, that is, the typing context at branches to the loop header will eventually match one of C_1, C_2, ..., C_N. The number of versions actually generated is expected to be low because the type of most variables remains stable for the duration of a function.

There is a risk of a combinatorial explosion when multiple versions of basic blocks are created eagerly. Consider the simple statement `x=a+b+c+d`. If the types of `a`, `b`, `c` and `d` are *unknown*, and those variables are live after the assignment, and there are two possible numerical types (`int` and `float`), there could be up to 16 versions of the basic block containing the assignment to `x`, one version for each set of type assignments to the variables being summed. In general, if basic block versioning is performed in an *eager* fashion and there are t possible types of values and a function has v variables, then there can be up to t^v versions of some basic blocks in that function. However, the logic of a program puts constraints on possible type combinations. In practice, not all the combinations of types are observed during an execution of a program.

It is often the case that variables are monomorphic in type (i.e. they always contain the same type of value). We can take advantage of this by *lazily* creating new block versions on demand. Versions for a particular context are only generated when that context is encountered during execution. *Lazy basic block versioning* doesn't completely eliminate the possibility of a combinatorial explosion in pathological cases, but this can be prevented by placing a hard limit on the number of versions generated for any given block. Some increase in code size is to be expected, but no more than a constant factor. Mueller and Whalley have shown [24] that specializing code through replication, while increasing the static size of machine code, can reduce the dynamic count of executed instructions and result in better cache usage.

Traditional type analyses often cannot infer a type for a variable, either because there is insufficient semantic information in the source program, or because the analysis is limited in its capabilities. For example, with an intraprocedural type analysis of JavaScript, no type information is known about function parameters. Without transforming the program, many variable types cannot be recovered by analysis alone. Moreover, the *unknown* type may propagate through primitive operations and effectively poison the results of such type analyses.

As will be demonstrated in Section 4, a key advantage of basic block versioning over program analyses lies in its ability to *recover unknown types*. The versioning approach is able to exploit type tests that are implicitly part of the language semantics to gain type information, and then generate new block versions where the additional type information remains known. Basic block versioning automatically unrolls some of the first iterations of loops in such a way that type tests are hoisted out of loop bodies. For example, if variables of *unknown* type are used unconditionally in a loop, their type will be tested only in the first iteration of the loop. The type information gained will allow further iterations to avoid redundant type tests.

Lazy basic block versioning bears some similarity to trace compilation [5] in the use of code duplication and type specialization to eliminate type tests [13]. Trace compilation typically relies on an interpreter to detect hot loops and record traces. It is also most effective on loop-heavy code. In contrast, lazy basic block versioning can handle any code structure just as effectively. It avoids the dual language implementation (interpreter and trace compiler) and requires no special infrastructure for profiling or recording traces.

The relative simplicity of tracking typing contexts and previously generated basic block versions means that the compiler avoids algorithms of high computational complexity. With a hard limit on the number of block versions, code generation time and code size scale linearly with the size of the input program. Lazy basic block versioning requires no external optimization or analysis passes to generate type-specialized code. This makes the approach interesting for use in baseline JIT compilers.

3 Implementation in Higgs

We have implemented lazy basic block versioning inside a JavaScript virtual machine called Higgs. This virtual machine comprises a JIT compiler targeted at x86-64 POSIX platforms. The current implementation of Higgs supports most of the ECMAScript 5 specification [18], with the exception of the `with` statement and the limitation that `eval` can only access global variables, not locals. Its runtime and standard libraries are self-hosted, written in an extended dialect of ECMAScript with low-level primitives. These low-level primitives are special instructions which allow expressing type tests as well as integer and floating point machine instructions in the source language.

In Higgs, functions are parsed into an abstract syntax tree and lazily compiled to a Static Single Assignment (SSA) Intermediate Representation (IR) when they are first called. Inlining is performed at this time according to simple fixed heuristic rules. Specific JavaScript runtime functions including arithmetic, comparison and object property access primitives are always inlined. This inlining allows exposing type tests and typed low-level operations contained inside primitives to the backend, which implements basic block versioning.

A basic block version corresponds to a basic block and an associated context containing type information about live values at the start of the block. Machine code generation always begins with the function's entry block and a default entry context being queued for compilation. Typing contexts in Higgs are implemented as sets of pairs associating live SSA values to unique type tags (see Section 3.2). Values for which no type information is known do not appear in the set. As each instruction in a block is compiled, information is both retrieved from and inserted into the current context. Information retrieved may be used to optimize the compilation of the current instruction (e.g. eliminate type tests). Instructions will also write their own output type in the context if known.

To guard against pathological cases where an unreasonably large number of versions would be generated, we have added one tunable parameter, `maxvers`, which specifies the maximum number of specialized versions that can be generated for any given basic block. Before the limit for a given block is reached, requests for new versions matching an incoming context will either find an existing exact match for the context, or compile a new version matching the incoming context exactly. Once the limit is reached for a particular block, requests for new versions of this block will first try to find an inexact but compatible match for the incoming context. An existing version is compatible with the incoming context if the value types assumed by the existing version are a subset of those specified in the incoming context.

```
/**
Context compatibility test function:
- Perfectly matching contexts produce score 0
- Imperfect matches produce a score > 0
- Incompatible matches produce Infinity
*/
Number contextComp(Context predCtx, Context succCtx)
{
    Number score = 0;

    // For each value live in the successor
    foreach (value in succCtx)
    {
        auto predType = predCtx.getType(value);
        auto succType = succCtx.getType(value);

        // If the successor has no known type,
        // we would lose a known type
        if (predType != UNKNOWN &&
            succType == UNKNOWN)
            score += 1

        // If the types do not match,
        // contexts are incompatible
        else if (predType != succType)
            return Infinity;
    }

    return score;
}
```

Figure 1 Context compatibility test function.

The context compatibility test is shown in Figure 1. A context containing less constraining types than the incoming context is compatible, but one that has more constraining types than the incoming context is not. Essentially, this allows for the loss of type information when transitioning along control flow edges. If the version limit was reached and no compatible match is found for a given block, a fully generic version of the block that assigns the *unknown* type to all live variables will be generated. This generic version is compatible with all possible incoming contexts. When the `maxvers` parameter is set to zero, basic block versioning is disabled, and only one generic version of each basic block may be generated.

3.1 Lazy Code Generation

Limiting the number of versions generated by *eager* basic block versioning to avoid combinatorial code growth is a difficult problem. Simply imposing a hard version limit is not a satisfactory solution because it is nontrivial to determine ahead of time which typing contexts are more probable than others, and which may not occur at all. This is particularly problematic in a JIT compiler, since compiling versions for type combinations that will not occur at run time translates into wasted compilation time, code bloat and poor usage of the instruction cache. There is also the issue of ordering machine code in memory so as to minimize the number of branches taken.

Clearly, basic block versioning ought to be guided by run time types, but gathering profiling data using traditional means could be expensive. Furthermore, the resulting data may be large and complex to analyze. Instead, Higgs delays the generation of block versions and lets the run time behavior of programs drive this process. The *execution of conditional branches* triggers the generation of new block versions. This is particularly useful since all

type tests are conditional branches. Versions are generated according to the types that actually occur at run time. This *lazy code generation* approach has four key benefits:

1. The order in which versions for different type combinations are generated tends to approximate the frequency of occurrence of the said types. This is particularly helpful in the presence of a block version limit.
2. It tends to produce efficient, cache-friendly linear orderings of compiled blocks in memory, as versions are generated in the order they are first executed.
3. Neither memory nor time are wasted compiling block versions for type combinations that never occur at run time. Type combinations that do not occur are never accounted for.
4. Unexecuted blocks are never compiled. Exception handling code is not generated for programs which do not throw exceptions. Floating point code is not generated for programs which do not make use of floating point values.

The Higgs backend lazily compiles versions of individual SSA basic blocks into x86-64 machine code as they are first executed. Higgs does not compile whole functions at once. Instead, the JIT compilation model employed by Higgs interleaves execution and compilation of basic blocks. The last instruction of a block, which must be a branch instruction, determines which block will be compiled next. If the branch is unconditional, or if its direction can be determined at compilation time, no branch instruction is generated, and the successor version the branch leads to is immediately compiled (unless already compiled, in which case a direct jump is written instead).

When a conditional branch whose direction cannot be determined at compilation time is encountered, a pair of out-of-line stubs are generated for the two possible outcomes of the branch, and execution resumes. Stubs, when executed, call back the compiler requesting compilation of the corresponding destination basic block with the typing context at the branch. The branch is then overwritten to fall through or jump to the generated basic block version. This way, the compilation of a particular basic block version is delayed until it is required for execution.

3.2 Type Tags and Runtime Primitives

Higgs segregates values into a few categories based on type tags [15]. These categories are: 32-bit integers (`int32`), 64-bit floating point values (`float64`), miscellaneous JavaScript constants (`const`), and four kinds of garbage-collected pointers inside the heap (`string`, `object`, `array`, `closure`). These type tags form a simple, first-degree notion of types that is used to drive the basic block versioning approach.

We chose this coarse-grained type classification to investigate the effectiveness and potential of basic block versioning. Higgs implements JavaScript operators as runtime library functions written in an extended dialect of JavaScript, and most of these functions use type tags to do type dispatching. As such, eliminating this first level of type tests as well as boxing and unboxing overhead, is crucial to improving the performance of the system as a whole.

Figure 2 illustrates the implementation of the `+` operator as an example. As can be seen, this function makes extensive use of low-level type test primitives such as `is_i32` and `is_f64` to implement dynamic dispatch based on the type tags of the input arguments. Most other arithmetic, comparison and property access primitives implement a similar dispatch mechanism.

Note that while according to the ES5 specification all JavaScript numbers are IEEE double-precision floating point values, high-performance JavaScript VMs typically attempt

```
function add(x, y) {
    if (is_i32(x)) { // If x is integer
        if (is_i32(y)) {
            if (var r = add_i32_ovf(x, y))
                return r;
            else // Handle the overflow case
                return add_f64(i32_to_f64(x),
                               i32_to_f64(y));
        } else if (is_f64(y))
            return add_f64(i32_to_f64(x), y);
    } else if (is_f64(x)) { // If x is fp
        if (is_i32(y))
            return add_f64(x, i32_to_f64(y));
        else if (is_f64(y))
            return add_f64(x, y);
    }

    // Eval args as strings and concat them
    return strcat(toString(x), toString(y));
}
```

Figure 2 Implementation of the + operator.

to represent small integer values using machine integers so as to improve performance by using lower latency integer arithmetic instructions. We have made the same design choice for Higgs. Consequently, JavaScript numbers are represented using tagged `int32` or `float64` values. Arithmetic operations on `int32` values may yield an `int32` or `float64` result, but arithmetic operations on `float64` values always yield an `float64` result.

3.3 Flow-based Representation Analysis

To provide a point of comparison and contrast the capabilities of basic block versioning with that of more traditional type analysis approaches, we have implemented a forward flow-based representation analysis that computes a fixed point on the types of SSA values. The analysis is an adaptation of Wegbreit's algorithm as described in [31]. It is an intraprocedural constant propagation analysis that propagates the types of SSA values in a flow-sensitive manner.

The representation analysis uses sets of possible type tags as a type representation. It is able to gain information from typed primitives (e.g. `add_f64` produces `float64` values) as well as type tests and forward this information along branches. The analysis is also able to deduce, in some cases, that specific branches will not be executed and ignore the effects of code that was determined dead. The type tags are the same as those used by basic block versioning, with the difference that basic block versioning only propagates unique known types and not type sets (e.g. `int32` ∪ `float64`). This means that basic block versioning can only propagate positive information gained from type tests whereas the analysis can propagate both positive and negative information (e.g. `a` is not `int32`).

We have chosen to give the type analysis a richer type representation than that of basic block versioning because several common arithmetic primitives can produce overflows that cannot be statically predicted. This means that most arithmetic operations can produce either `int32` or `float64` types. If the type analysis could not represent this type set, it would be forced to infer that the output type of most arithmetic operations is of *unknown* type. This would immediately put the type analysis at an enormous disadvantage when compared to basic block versioning because basic block versioning is not affected by overflows that do not occur at run time.

```
function sum(n) {
    for (var i=0, s=0; i<n; i++)
        s += i;
    return s;
}
```

Figure 3 The `sum` function.

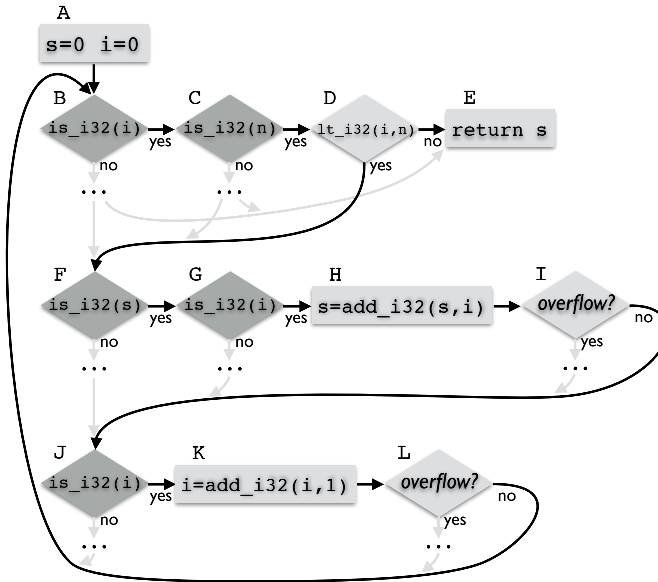


Figure 4 Control flow graph of `sum` function (unexecuted parts omitted).

3.4 Concrete Example

To illustrate the lazy basic block versioning approach, we will explain the compilation of the `sum` function given in Figure 3. Specifically, we trace the execution of the function call `sum(500)`. This only requires 32-bit integer computations because no overflows occur for this value of `n`. Figure 4 shows the parts of the control flow graph of the function executed during this call. The complete graph is larger and includes code to handle floating point values and other types. Unexecuted parts of the control flow graph are shown as ellipses (...).

Before versioning, there are 5 type tests on `i` and `n` executed as part of the loop. Higgs compiles the code for the `sum` function incrementally, type-specializing and eliminating type tests as compilation proceeds. The compiled and specialized code is equivalent to the control flow graph shown in Figure 5. Multiple blocks have been specialized based on the knowledge that `i`, `s` and `n` are of the `int32` type. Only one type test is left, in block `C1`, and this type test has been hoisted out of the loop. It is executed only once per call to `sum`.

The incremental compilation process occurs in six steps and is illustrated in Figure 6. When first entering the `sum` function, a version of the entry block `A` is compiled, generating `A1`. Variables `s` and `i` are initialized to `int32` and this is noted in the current typing context. Then, block `B` is compiled down to nothing because `i` is known to be `int32` in the current context. In `C1`, generated from block `C`, the type test on `n` needs to emit machine code because the type of `n` is *unknown* in the current context and so must be tested. Therefore, stubs `stub_n_not_i32` and `stub_D1` are generated and execution resumes at `A1`.

Because `n` contains an `int32`, execution flows to `stub_D1`, which calls back into the JIT compiler. The branch instructions at the end of block `C1` is rewritten so that a jump to a

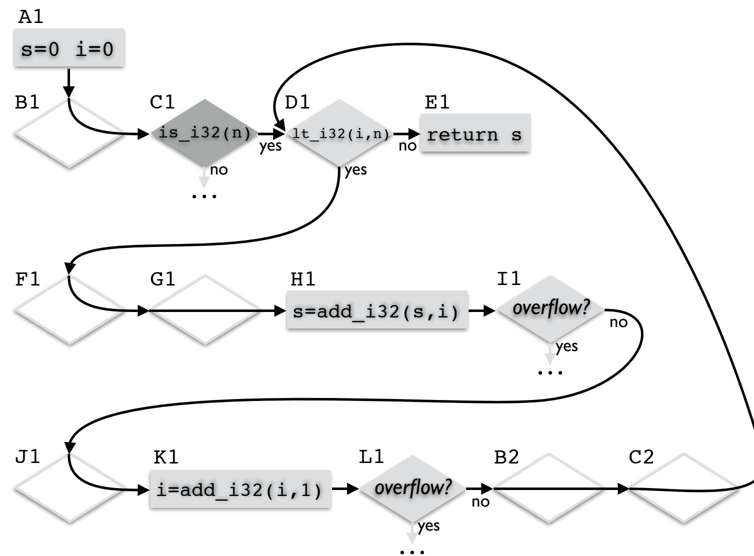


Figure 5 Control flow graph of `sum` function transformed by basic block versioning.

stub is executed only if `n` is not `int32`. In future calls of `sum` where `n` is `int32`, the branch will fall through to block `D1`. The generation of block `D1` from `D` is handled similarly. Two stubs (`stub_F1` and `stub_E1`) are used to determine the direction of the less-than comparison branch, which is unknown at compilation time. Execution then resumes at `D1` and flows to `stub_F1`. This time, the JIT compiler inverts the direction of the branches at the end of block `D1` so that the fall through will be block `F1`. Then blocks `F1`, `G1`, `H1`, and `I1` are generated and execution resumes at `F1`.

The code is incrementally generated in this fashion by successively executing `stub_J1`, `stub_B2`, and `stub_E1`. After the execution of `stub_B2`, the emitted code executes until the end of the loop. In the last loop iteration, the less-than comparison in `D1` fails. This triggers compilation of the loop exit block `E1`, which is conveniently placed outside of the loop body. We note that the detailed sequence of instructions needed to return from `sum` is more complex than what is shown (to support JavaScript's variable arity function calls).

The right part of Figure 6 shows the generated code after the execution of `sum(500)` has completed. Type tests in blocks `F1`, `G1` and `J1` were eliminated because `i`, `s` and `n` are known to be `int32` at those points. The jump back to the loop header in `L1` generated new versions of blocks `B` and `C` where `i`, `s` and `n` are known to be `int32`. Hence, only the first loop iteration performs a type test.

4 Evaluation

4.1 Experimental Setup

To assess the effectiveness of basic block versioning, we have tested it on a total of 26 classic benchmarks from the SunSpider and Google V8 suites. One benchmark from the SunSpider suite and one from the V8 suite were not included in our tests because Higgs does not yet implement the required features. Benchmarks making use of regular expressions were discarded because unlike V8 and TraceMonkey, Higgs does not implement JIT compilation of regular expressions, and neither does Truffle JS [33, 32].

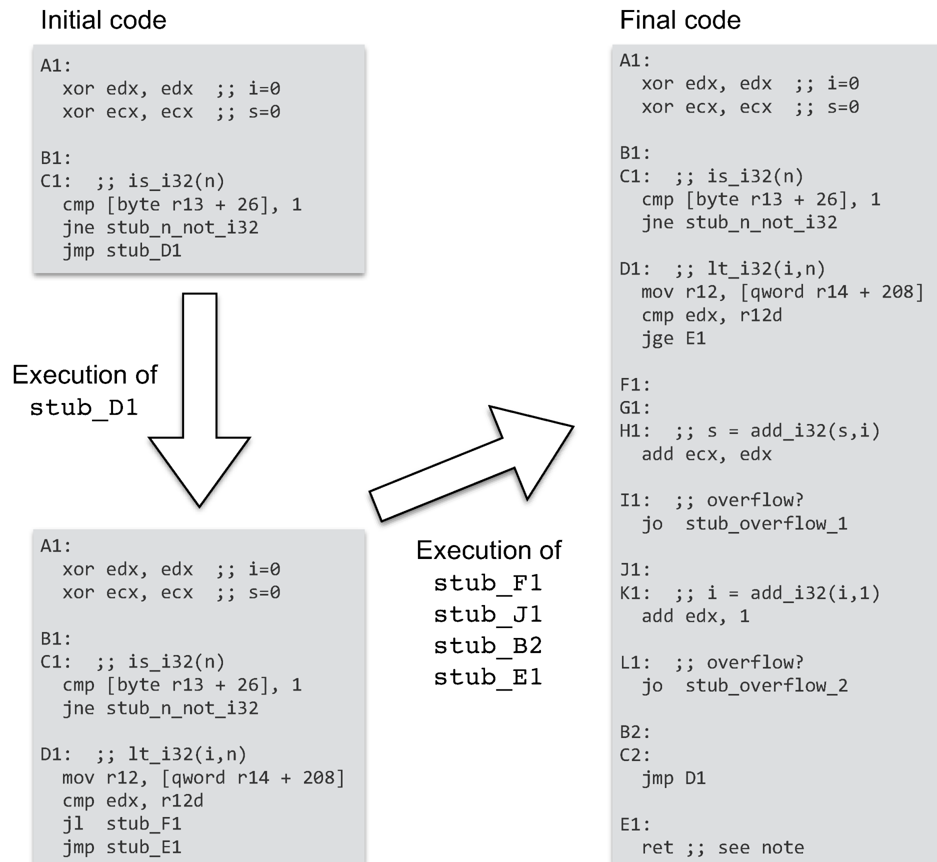


Figure 6 Machine code at different steps of code compilation.

Since Higgs interleaves compilation and execution and which parts of a program are eventually compiled is entirely dependent on run time behavior, we have measured approximate compilation times using a microsecond counter which is started and stopped when compilation begins and ends. The total times accumulated are averaged across 10 runs to give a final compilation time figure.

To measure execution time separately from compilation time in a manner compatible with V8, TraceMonkey, Truffle JS and Higgs, we have modified benchmarks so that they could be run in a loop. A number of warmup iterations are first performed so as to trigger JIT compilation and optimization of code before timing runs take place.

The number of warmup and timing iterations were scaled so that short-running benchmarks would execute for at least 1000ms in total during both warmup and timing. Unless otherwise specified, all benchmarks were run for at least 10 warmup iterations and 10 timing iterations.

V8 version 3.29.66, TraceMonkey version 1.8.5+ and Truffle JS v0.5 were used for performance comparisons. Tests were executed on a system equipped with an Intel Core i7-4771 quad-core CPU with 8MB L3 cache and 16GB of RAM running Ubuntu Linux 12.04. Dynamic CPU frequency scaling was disabled for our experiments.

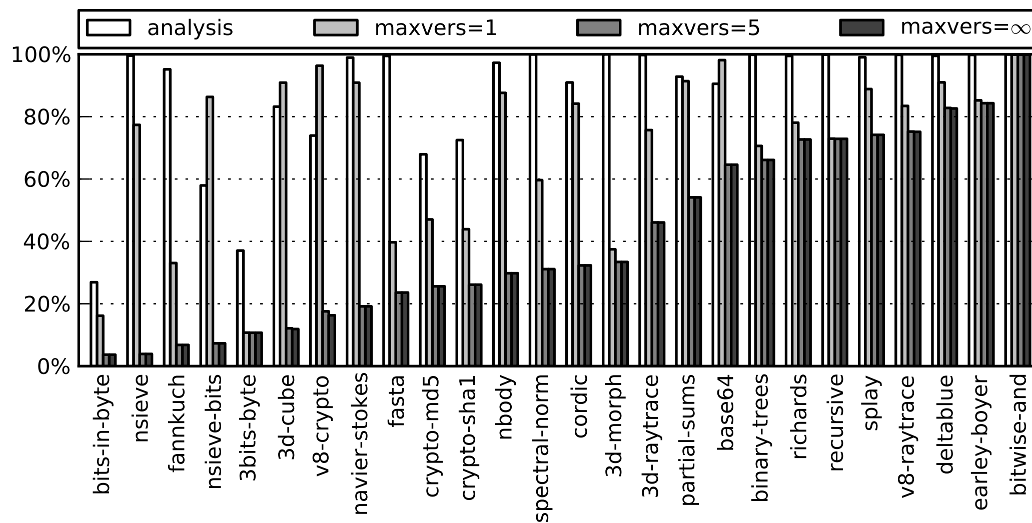


Figure 7 Dynamic counts of type tests executed using the representation analysis and lazy basic block versioning with various version limits (relative to baseline).

4.2 Dynamic Type Tests

Figure 7 shows the dynamic counts of type tests for the representation analysis and for lazy basic block versioning with various block version limits. These counts are relative to a baseline which has the version limit set to 0, and thus only generates a generic version of each basic block. As can be seen from the counts, the analysis produces a reduction in the number of dynamically executed type tests over the unoptimized baseline on most benchmarks. The basic block versioning approach does at least as well as the analysis, and almost always significantly better. Surprisingly, even with a version cap as low as 1 version per basic block, the versioning approach is often competitive with the representation analysis. This is likely because most value types are monomorphic.

Raising the version cap reduces the number of type tests performed with the versioning approach in an asymptotic manner as we get closer to the limit of what is achievable with our implementation. The versioning approach does quite well on the `bits-in-byte` benchmark. This benchmark (see Figure 8) is an ideal use case for our versioning approach. It is a tight loop performing bitwise and arithmetic operations on integers which are all stored in local variables. The versioning approach performs noticeably better than the analysis on this test because it is able to test the type of the function parameter `b`, which is initially unknown when entering `bitsinbyte` only once per function call and propagate this type thereafter. The analysis on its own cannot achieve this, and so must repeat the test for each operation on `b`. In contrast, the `bitwise-and` benchmark operates exclusively on global variables, for which our system cannot extract types, and so neither the type analysis nor the versioning approach show any improvement over baseline for this benchmark.

A breakdown of relative type test counts by kind, averaged across all benchmarks (using the geometric mean) is shown in Figure 9. We see that the versioning approach is able to perform as well or better than the representation analysis across each kind of type test. The `is_closure` category shows the least improvement. This is because functions are typically globals or methods, which basic block versioning cannot yet get type information about. We note that versioning is much more effective than the analysis when it comes to eliminating `is_i32` type tests. This is because integer and floating point types often get intermixed,

```
function bitsinbyte(b) {
    var m = 1, c = 0;
    while(m < 0x100) {
        if(b & m) c++;
        m <<= 1;
    }
    return c;
}

function TimeFunc(func) {
    var x, y, t;
    for(var x=0; x<350; x++)
        for(var y=0; y<256; y++) func(y);
}

TimeFunc(bitsinbyte);
```

Figure 8 SunSpider bits-in-byte benchmark.

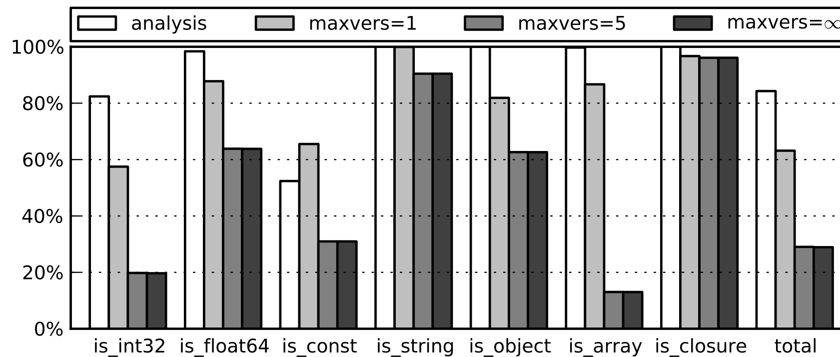


Figure 9 Type test counts by kind of type test (relative to baseline).

leading to cases where the analysis cannot eliminate such tests. The versioning approach has the advantage that it can replicate and detangle integer and floating point code paths. A limit of 5 versions per block eliminates 71% of total type tests, compared to 16% for the analysis.

4.3 Code Size Growth

Figure 10 shows the relative proportion of blocks for which different counts of versions were generated across all benchmarks. As one might expect, the relative proportion of blocks tends to steadily decrease as the number of versions is increased. Most basic blocks have only one version, 5.2% have two, and just 0.16% of blocks have 5 versions or more. Hence, blocks with a large number of versions are a rare occurrence.

The maximum number of versions ever produced for a given block across our benchmarks is 11. This occurs in the `v8-raytrace` benchmark. The function generating the most block versions in this benchmark is `rayTrace`. This function is at the core of the ray tracing algorithm. It contains a loop with several live variables used during iteration. Some of these variables can be either `null` or an object reference. There are also versions generated where basic block versioning cannot determine a type for some variables.

The effects of basic block versioning on the total generated code size are shown in Figure 11. It is interesting to note that the representation analysis almost always results in a

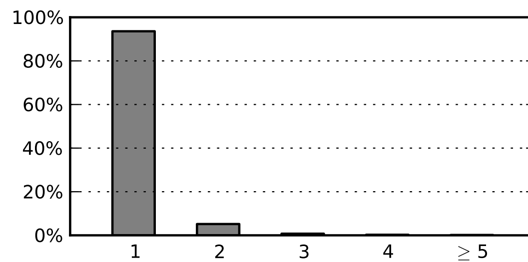


Figure 10 Relative occurrence of block version counts.

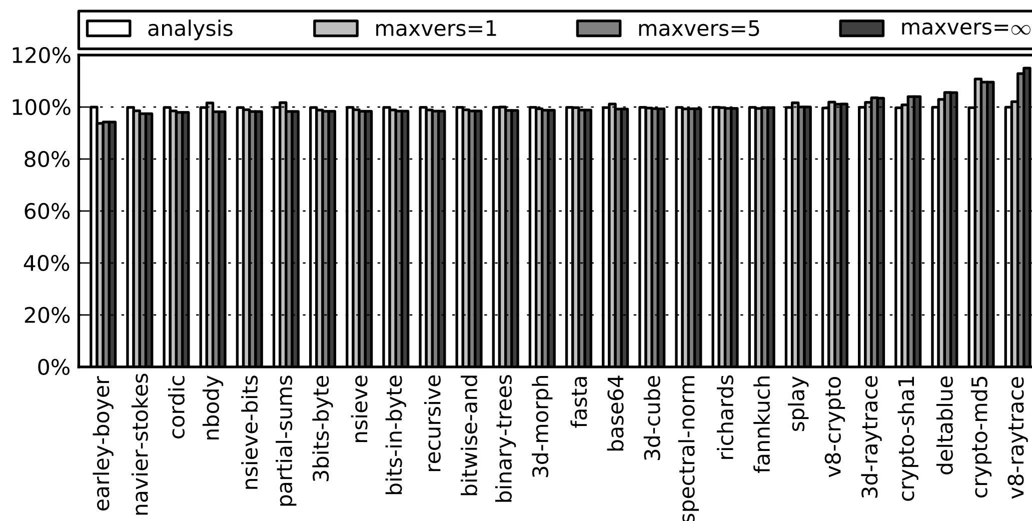


Figure 11 Code size for various block version limits (relative to baseline).

slight reduction in code size. This is because the analysis allows the elimination of type tests and the generation of more optimized code, which is usually smaller. On the other hand, basic block versioning can generate multiple versions of basic blocks, which often (but not always) results in more generated code. The volume of generated code does not increase linearly with the block version limit. Rather, it tapers off as a limited number of versions tends to be generated for each block. A limit of 5 versions per block results in a mean code size increase of 0.19%. With no limit at all on the number of versions, the code size increase does not change much, with a mean of 0.25% and a maximum increase of 15% across all benchmarks. On the benchmarks we have tested, there is no pathological code size explosion, and the block version limit is not strictly necessary.

4.4 Execution Time

Figure 12 shows the execution times relative to baseline. Because our type analysis is not optimized for speed and incurs a significant compilation time penalty, we have excluded compilation time and measured only time spent executing compiled machine code. A limit of 5 versions per block produces on average a 21% reduction in execution time, and speedups of up to 1.2%, while the type analysis yields a 4% average speedup.

In most cases, basic block versioning produces a notable reduction in relative execution time that compares favorably with the static analysis. The intraprocedural type analysis

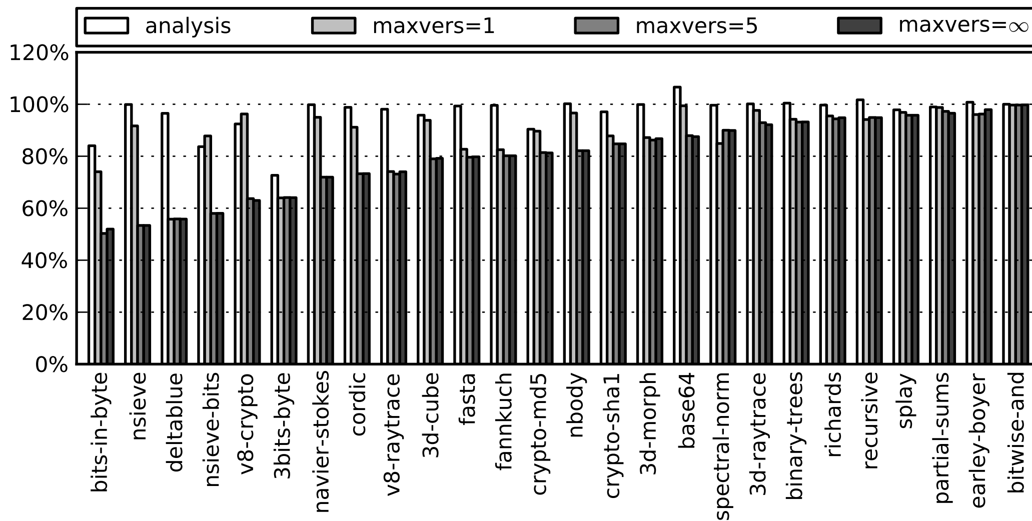


Figure 12 Execution time for various block version limits (relative to baseline).

does not eliminate enough type tests to be effective in improving execution times. We believe that it should be possible to significantly improve upon the basic block versioning results with method inlining and better optimized property accesses, which would expose more type tests and more precise type information.

4.5 Eager Versioning

In order to evaluate the importance of lazyness in our basic block versioning approach, we have tested an older version of Higgs which generates block versions eagerly. In this configuration, whole methods are compiled at once, never producing stubs, and specialized versions are generated for a given block until the block version limit is hit. The versions are generated in no particular order. The performance obtained with eager generation of block versions was found to be inferior on all metrics. When the version limit is set to 5, on average, the eager approach eliminates about half as many type tests as the lazy approach, the code size is 223% of baseline on average (see Figure 13), and the execution time is 5% slower than baseline.

There are multiple issues with the eager generation of block versions. The most important one is that without some form of lazyness, without code stubs, we must always produce code for both sides of a conditional branch. In the case of eager basic block versioning, this means we generate code for both branches of a type test, even though in most cases only one side of the branch is ever taken. We end up generating versions for a large number of type combinations which cannot occur at run time, but which we have no heuristic to discard at method compilation time. The number of possible type combinations increases exponentially with the number of live variables, and so the block version limit is rapidly reached. Since versions are generated in no particular order, the specialized versions eagerly generated before the block version limit is hit are likely to be versions matching irrelevant type combinations.

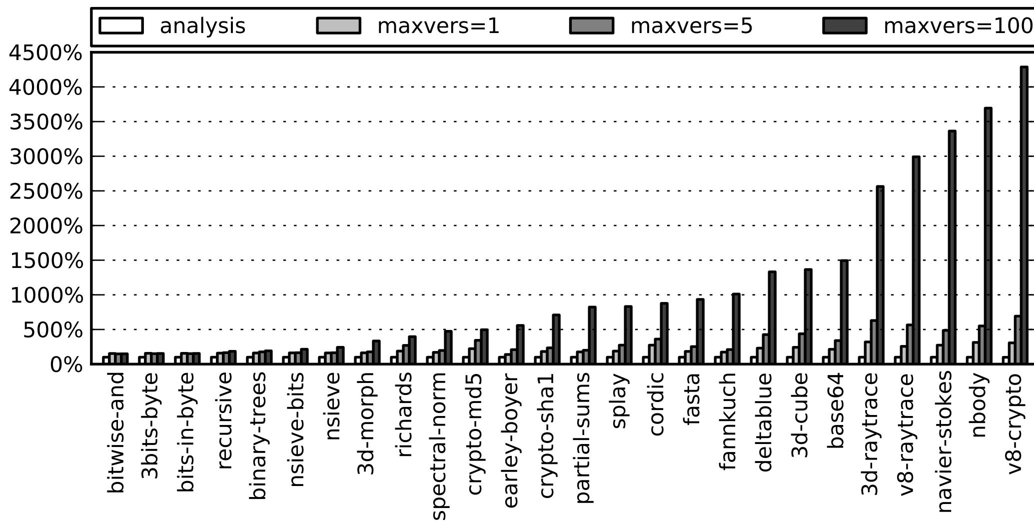


Figure 13 Code size with eager basic block versioning (relative to baseline).

4.6 Compilation Time

The graph in Figure 14 shows a comparison of the total compilation time with the type analysis and with different block version limits relative to baseline. The type analysis, as implemented, is not particularly efficient because it passes around maps of SSA values to type sets and iterates until a fixed point is reached. This is expensive and scales poorly with program size. The analysis increases compilation time by over 100% in many cases. In the worst case, on the `earley-boyer` benchmark, the analysis incurs a compilation time slowdown of more than 100 times.

Basic block versioning does not increase compilation times by much. A limit of 5 versions per block produces a compilation time decrease of 1.2% on average, and a 5.3%increase in the worst case. It is interesting to note that in many cases, enabling basic block versioning reduces compilation time by a small amount. This is because specializing code to eliminate type checks often makes it smaller, and for some basic blocks, no machine code is generated at all.

4.7 Comparison against the V8 Baseline Compiler

We have compared the execution time of the machine code generated by Higgs to that of the V8 baseline compiler. The V8 baseline compiler is not to be confused with Crankshaft. It is a low-overhead method-based JIT which, like Higgs, does not perform method inlining and only performs basic optimizations and fast on-the-fly register allocation. It is meant to compile code rapidly.

Figure 15 shows speedups of Higgs over V8 baseline. The scale is logarithmic, and higher bars indicate better performance on the part of Higgs. As can be seen, Higgs delivers better performance on more than half of the benchmarks. The three benchmarks on which V8 baseline does best are from the V8 suite, which the V8 baseline compiler was tailored to perform best on. Higgs is able to deliver impressive speedups on a variety of benchmarks in various areas of interest including floating point arithmetic, object-oriented data structures and string manipulation.

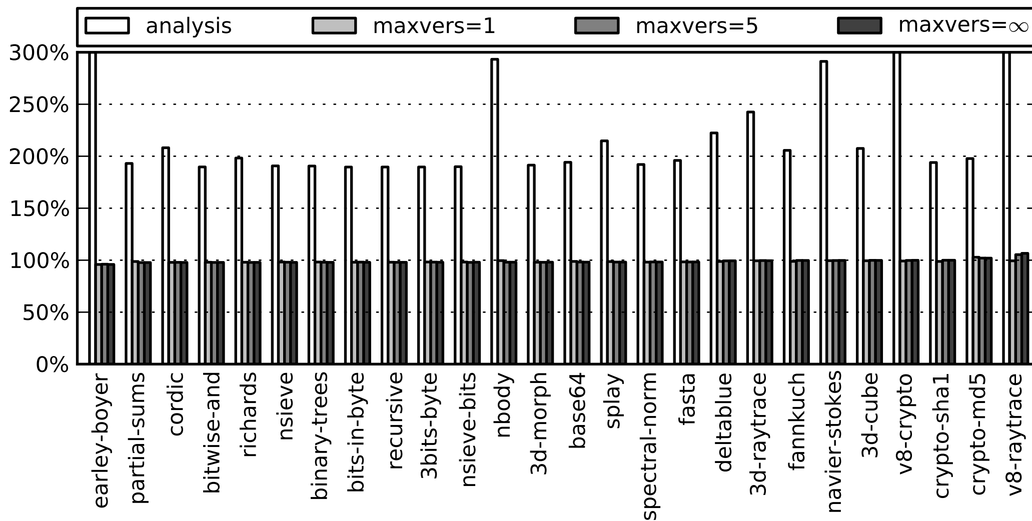


Figure 14 Compilation time for various block version limits (relative to baseline).

4.8 Comparison against TraceMonkey

The similarity of trace compilation and basic block versioning has prompted us to compare Higgs to TraceMonkey, a tracing JIT compiler for JavaScript that was part of Mozilla's SpiderMonkey until mid 2011. It has the ability to eliminate type checks [13] based on analysis of traces. Note that Higgs does not yet implement inlining of method calls whereas TraceMonkey can inline them as part of tracing.

Figure 16 shows speedups of Higgs over TraceMonkey. The scale is again logarithmic, with higher bars indicating better performance on the part of Higgs. TraceMonkey performs better on many benchmarks. Unsurprisingly, the benchmarks TraceMonkey achieves the best performance on tend to be benchmarks which include short and predictable loops. In these, TraceMonkey is presumably able to inline all function calls which puts Higgs, without inlining, at a significant performance disadvantage.

It is interesting that Higgs, even without inlining, does much better on some of the largest benchmarks from our set. The two raytrace benchmarks, for example, 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.

Higgs shines in benchmarks with complex, unpredictable control flow as well as recursive computations. TraceMonkey is in no way the pinnacle of tracing JIT technology, but there are clearly areas where basic block versioning unambiguously wins over this implementation of trace compilation. Whereas tracing, in its simplest forms, is ideal for predictable loops, basic block versioning is not biased for any kind of control-flow structures. We believe that implementing inlining in Higgs would likely even the performance gap on the benchmarks where Higgs currently performs worse.

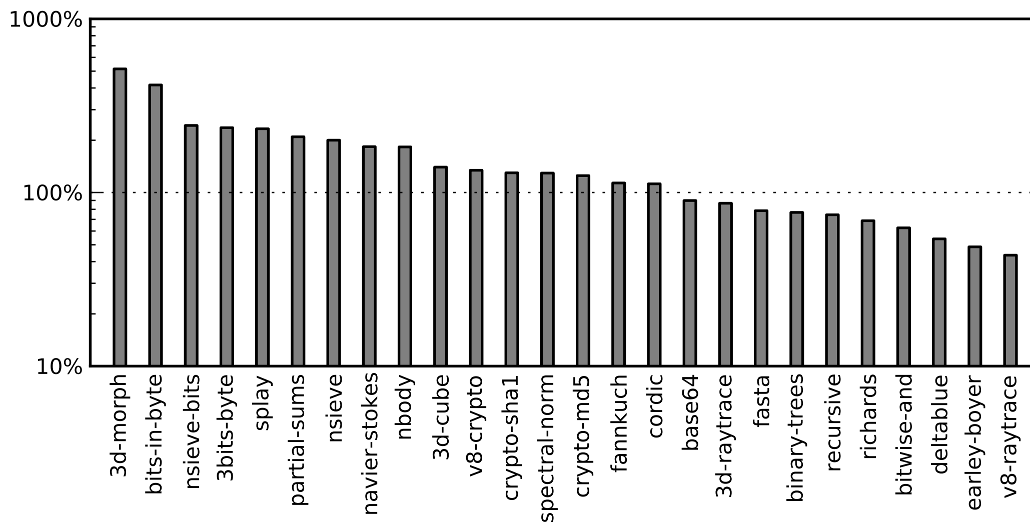


Figure 15 Speedup relative to V8 baseline (log scale, higher is better).

4.9 Comparison against Truffle JS

Figure 17 shows the relative speed of Higgs over Truffle JS on a logarithmic scale, with higher bars indicating better performance on the part of Higgs. We have evaluated the performance with 1, 10 and 100 warmup iterations. With 1 or 10 warmup iterations, Higgs outperforms Truffle on the majority of benchmarks, with speedups of up to 30x in some cases.

With 100 warmup iterations, the picture changes, and Truffle outperforms Higgs on most benchmarks. This seems to be because Truffle interprets code for a long time before compiling and optimizing it. In contrast, Higgs only needs to compile and execute a given code path once before it is optimized, with no warmup executions required.

Truffle has two main performance advantages over Higgs. The first is that after warmup, Truffle is able to perform deep inlining, as illustrated by the `v8-raytrace` benchmark. The second is that Truffle has sophisticated analyses which Higgs does not have. For instance, the recorded time for the `3bits-byte` microbenchmark is zero, suggesting that Truffle was able to entirely eliminate the computation performed as its output is never used. Doing this requires a side-effect analysis which can cope with the semantic complexities of JavaScript.

We note that even with 100 warmup iterations, and despite Truffle's powerful optimization capabilities, there remain several benchmarks where Higgs performs best, with speedups over 10x in some cases.

5 Related Work

The *tracelet-based* approach used by Facebook's HipHop VM for PHP (HHVM) [1] bears much similarity to our own. It is based on the JIT compilation of small code regions (tracelets) which are single-entry multiple-exit basic blocks. Each tracelet is type-specialized based on variable types observed at JIT 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 assumptions and chained to the failing guards.

There are three important differences between the HHVM approach and basic block versioning. The first is that BBV does not insert dynamic guards but instead exposes and

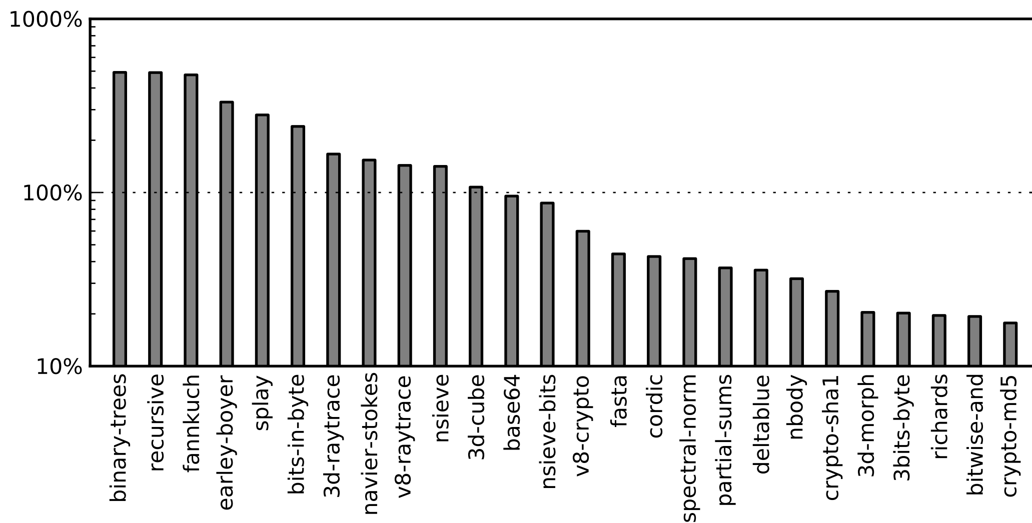


Figure 16 Speedup relative to TraceMonkey (log scale, higher is better).

exploits the underlying type checks that are part of the definition of runtime primitives. HHVM cannot do this as it uses monolithic high-level instructions to represent PHP primitives, whereas the Higgs primitives are self-hosted and defined in an extended JavaScript dialect. The second difference is that BBV propagates known types to successors and doesn't usually need to re-check the types of local variables. A third important difference is that HHVM uses an interpreter as fallback when too many tracelet versions are generated. Higgs falls back to generic basic block versions which do not make type assumptions but are still always JIT compiled for better performance.

Trace compilation, originally introduced by the Dynamo [5] native code optimization system, and later applied to JIT compilation in HotpathVM [14] aims to record long sequences of instructions executed inside hot loops. Such linear sequences of instructions often make optimization simpler. Type information can be accumulated along traces and used to specialize code and remove type tests [13], overflow checks [28] or unnecessary allocations [7]. 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 JIT. Code is also compiled lazily, as needed, without compiling whole functions at once.

The simplicity of basic block versioning is one of its main advantages. It does not require external infrastructure such as an interpreter to execute code or 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 is a large number of control flow paths going through a loop. 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. 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.

Run time type feedback uses profiling to gather type information at execution time. This information is then used to optimize dynamic dispatch [17]. There are similarities with basic block versioning, which generates optimized code paths lazily based on types occurring at run time. The two techniques are complementary. Basic block versioning could be made more efficient by using type profiling to reorder sequences of type tests in a type dispatch.

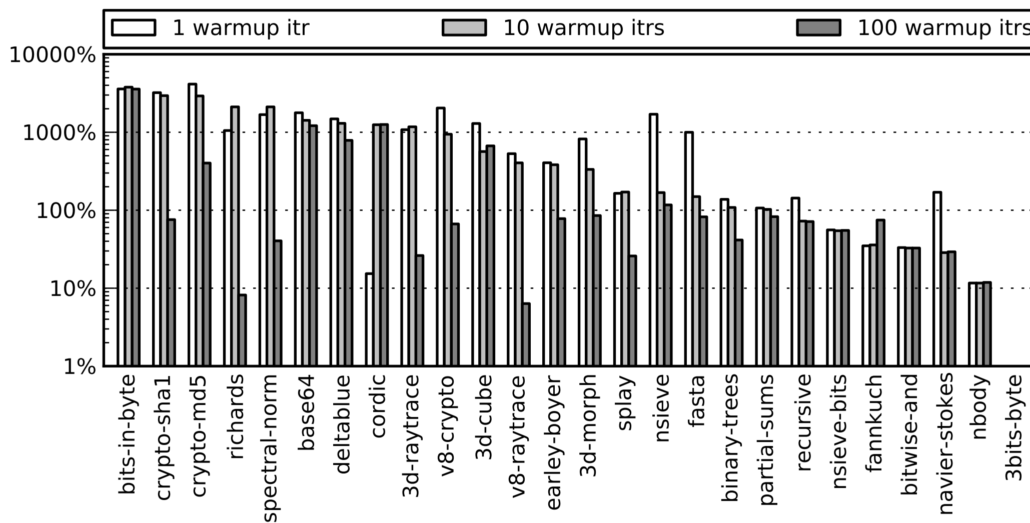


Figure 17 Speedup relative to Truffle JS (log scale, higher is better).

Type feedback could be augmented by using basic block versioning to generate multiple optimized code paths. The Truffle framework uses run time type feedback combined with guards to type-specialize AST nodes at run time [33, 32].

There have been multiple efforts to devise type analyses for dynamic languages. The Rapid Atomic Type Analysis (RATA) [22] is an intraprocedural flow-sensitive analysis based on abstract interpretation that aims to assign unique types to each variable inside of a function. Attempts have also been made to define formal semantics for a subset of dynamic languages such as JavaScript [4], Ruby [12] and Python [3], 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 [19][20][21]. Such analyses are usable in the context of static code analysis, but take too long to execute to be usable in VMs and do not deal with the complexities of dynamic code loading.

More recently, work done by Brian Hackett et al. at Mozilla resulted in an interprocedural hybrid type analysis for JavaScript suitable for use in production JIT compilers [16]. This analysis represents an important step forward for dynamic languages, but as with other type analyses, must conservatively assign one type to each value, making it 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.

Basic block versioning bears some similarities to classic compiler optimizations such as *loop unrolling* [11], *loop peeling* [29], and *tail duplication*, considering it achieves some of the same results. Another parallel can be drawn with *Partial Redundancy Elimination (PRE)* [23]; the versioning approach seeks to eliminate and hoist out of loops a specific kind of redundant computation: dynamic type tests. *Code replication* has also been used to improve the effectiveness of PRE [6].

Basic block versioning is also similar to the idea of *node splitting* [30]. This technique is an analysis device designed to make control flow graphs reducible and more amenable to analysis. The *path splitting* algorithm implemented in the SUIF compiler [27] aims at improving reaching definition information by replicating control flow nodes in loops to

eliminate joins. Unlike basic block versioning, these algorithms cannot gain information from type tests. The algorithms as presented are also specifically targeted at loops, while basic block versioning makes no special distinction. Mueller and Whalley have developed effective static analyses that use *code replication* to eliminate both unconditional and conditional branches [24][25]. However, their approach is intended to optimize loops and operates on a low-level intermediate representation that is not ideally suited to the elimination of type tests in a high-level dynamic language.

Customization is a technique developed to optimize Self programs [8] that compiles multiple copies of methods specialized on the receiver object type. Similarly, *type-directed cloning* [26] 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 [10] and similar work done for the MaJIC MATLAB compiler [2] 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.

Basic block versioning also resembles the *iterative type analysis* and *extended message splitting* techniques developed for Self by Craig Chambers and David Ungar [9]. This is a combined static analysis and 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. Basic block versioning is simpler to implement and generates code lazily, requiring less compilation time and memory overhead, making it more suitable for integration into a baseline JIT compiler.

6 Limitations and Future Work

Since Higgs is a standalone JavaScript VM that is not integrated in a web browser, we have tested it on out-of-browser benchmarks that are most relevant to using JavaScript in the server-side space (like node.js[1]). We do not anticipate any issues with using basic block versioning in a JavaScript VM integrated into a web browser, but we have not done the integration required for such an experiment. Basic block versioning is suitable for optimizing dynamic languages in general, not just JavaScript web applications in particular.

Several extensions to basic block versioning are possible. For instance, we have successfully extended it to perform overflow check elimination on loop increments, but have kept this feature disabled to simplify the presentation in this paper. Another interesting extension of basic block versioning would be to propagate information about global variable types, object identity and object property types. It may also be desirable to know the exact value of some variables and object fields, particularly for values likely to remain constant.

The implementation of lazy basic block versioning evaluated in this paper only tracks type information intraprocedurally. It would be beneficial to apply basic block versioning to function calls so that type information can propagate from caller to callee. This would entail

[1] `http://nodejs.org`

having multiple specialized entry points for parameter types encountered at the call sites of a function. Similarly, call continuation blocks (return points) could be versioned to allow information about return value types to flow back to the caller.

7 Conclusion

We have described a simple approach to JIT compilation called lazy basic block versioning. This technique combines code generation with type propagation and code duplication to produce more optimized code through the accumulation of type information during code generation. The versioning approach is able to perform optimizations such as automatic hoisting of type tests and efficiently detangles code paths along which multiple numerical types can occur. Our experiments show that in most cases, basic block versioning eliminates significantly more dynamic type tests than is possible using a traditional flow-based type analysis. It eliminates up to 71% of type tests on average with a limit of 5 versions per block, compared to 16% for the analysis we have tested, and never performs worse than such an analysis.

We have empirically demonstrated that although our implementation of basic block versioning does increase code size in some cases, the resulting increase is quite small and pathological code size explosions are unlikely to occur. In our experiments, a limit of 5 versions per block results in a mean code size increase of just 0.19%. Our experiments with Higgs also indicate that lazy basic block versioning improves performance up to 1.2% with a limit of 5 versions per block. Finally, we have shown that Higgs performs better than the V8 baseline compiler on most of our benchmarks, and better than TraceMonkey on several of the more complex benchmarks in our set.

Basic block versioning is a simple and practical technique that requires little implementation effort and offers important advantages in JIT-compiled environments where type analysis is often difficult and costly. Dynamic languages, which perform a large number of dynamic type tests, stand to benefit the most.

Higgs is open source and the code used in preparing this publication is available on GitHub[2].

Acknowledgements. Special thanks go to Paul Khuong, Laurie Hendren, Erick Lavoie, Tommy Everett, Brett Fraley and all those who have contributed to the development of Higgs.

This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Mozilla Corporation.

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 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.

[2] `https://github.com/higgsjs/Higgs/tree/ecoop2015`

3 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.

4 Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *Proceedings of ECOOP 2005*, pages 428–452. Springer Berlin Heidelberg, 2005.

5 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.

6 R. Bodík, R. Gupta, and M. L. Soffa. Complete removal of redundant expressions. In *Proceedings of the 1998 conference on Programming Language Design and Implementation (PLDI)*, pages 1–14. ACM New York, 1998.

7 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.

8 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.

9 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.

10 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.

11 Jack W Davidson and Sanjay Jinturkar. Aggressive loop unrolling in a retargetable, optimizing compiler. In *Proceedings of the 1996 international conference on Compiler Construction (CC)*, pages 59–73. Springer Berlin Heidelberg, 1996.

12 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.

13 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. In *Proceedings of the 2009 conference on Programming Language Design and Implementation (PLDI)*, pages 465–478. ACM New York, 2009.

14 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.

15 David Gudeman. Representing type information in dynamically typed languages, 1993.

16 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.

17 Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the 1994 conference on Programming Language Design and Implementation (PLDI)*, pages 326–336. ACM New York, 1994.

18 ECMA International. *ECMA-262: ECMAScript Language Specification.* European Association for Standardizing Information and Communication Systems (ECMA), Geneva, Switzerland, fifth edition, 2009.

19 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.

20 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.

21 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.

22 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.

23 E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, February 1979.

24 Frank Mueller and David B. Whalley. Avoiding unconditional jumps by code replication. In *Proceedings of the 1992 conference on Programming Language Design and Implementation (PLDI)*, pages 322–330. ACM New York, 1992.

25 Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the 1995 conference on Programming Language Design and Implementation (PLDI)*, pages 56–66. ACM New York, 1995.

26 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.

27 Massimiliano Antonio Poletto. *Path splitting: a technique for improving data flow analysis.* PhD thesis, MIT Laboratory for Computer Science, 1995.

28 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.

29 Litong Song and Krishna M Kavi. A technique for variable dependence driven loop peeling. In *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, pages 390–395. IEEE, 2002.

30 Sebastian Unger and Frank Mueller. Handling irreducible loops: optimized node splitting versus dj-graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):299–333, July 2002.

31 Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, April 1991.

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 *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204. ACM New York, 2013.

33 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.

Loop Tiling in the Presence of Exceptions

Abhilash Bhandari and V. Krishna Nandivada

Department of CSE, IIT Madras, Chennai, India
`abilash@cse.iitm.ac.in`, `nvk@cse.iitm.ac.in`

Abstract

Exceptions in OO languages provide a convenient mechanism to deal with anomalous situations. However, many of the loop optimization techniques cannot be applied in the presence of conditional `throw` statements in the body of the loop, owing to possible cross iteration control dependences. Compilers either ignore such `throw` statements and apply traditional loop optimizations (semantic non-preserving), or conservatively avoid invoking any of these optimizations altogether (inefficient). We define a loop optimization to be *exception-safe*, if the optimization can be applied even on (possibly) exception throwing loops, in a semantics preserving manner. In this paper, we present a generalized scheme to do exception-safe loop optimizations and present a scheme of optimized exception-safe loop tiling (oESLT), as a specialization thereof.

oESLT tiles the input loops, assuming that exceptions will never be thrown. To ensure the semantics preservation (in case an exception is thrown), oESLT generates code to rollback the updates done in the *advanced* iterations (iterations that the unoptimized code would not have executed, but executed speculatively by the oESLT generated code) and safely-execute the *delayed* iterations (ones that the unoptimized code would have executed, but not executed by the code generated by oESLT). For the rollback phase to work efficiently, oESLT identifies a minimal number of elements to backup and generates the necessary code. We implement oESLT, along with a naive scheme (nESLT, where we backup every element and do a full rollback and safe-execution in case an exception is thrown), in the Graphite framework of GCC 4.8. To help in this process, we define a new program region called ESCoPs (Extended Static Control Parts) that helps identify loops with multiple exit points and interface with the underlying polyhedral representation. We use the popular PolyBench suite to present a comparative evaluation of nESLT and oESLT against the unoptimized versions.

1998 ACM Subject Classification D.3.4 [Programming Languages] Processors – Optimization, Compilers

Keywords and phrases Compiler optimizations, semantics preservation, exceptions, loop-tiling

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.124

1 Introduction

Exceptions are one of the useful features in modern languages such as C++, Java, C#, ML and so on. Exceptions provide a structured way to handle anomalous and unexpected behaviors in the program and are finding increasing use in real world applications. While exceptions improve the programmability aspects, they have an impact on the generation of efficient code. The presence of exception throwing statements (explicitly in C++, Java, C#, ML – using a `throw` statement, or implicitly in C# and Java – for example, ArrayIndexOutOfBounds Exception) in the programs work as a deterrent to many compiler optimizations and analyses. This is because of the additional control flow edges and dependences (which cannot be resolved statically) that get introduced due to the presence of exceptions. We will illustrate the same using an example.

© Abhilash Bhandari and V. Krishna Nandivada;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 124–148

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
try{
 for(i=0;i<M;i++){
  for(j=0;j<N;j++){
    if(cond){
     throw e; }
    a[i][j]=b[j][i];
  }
 }
}catch(Ex e){ .. }
```

(a) Matrix transpose.

```
try{
 for(ii=0;ii<M/bn;ii++){
  for(jj=0;jj<N/bn;jj++){
   for(i=ii*bn;i<min(M,(ii+1)*bn);i++){
    for(j=jj*bn;j<min(N,(jj+1)*bn);j++){
      if(cond){throw e;}
      a[i][j]=b[j][i];
    } } } }
}catch(Ex e){ .. }
```

(b) An incorrectly tiled matrix transpose.

Figure 1 Matrix transpose and incorrect tiling.

Figure 1a shows the snippet of a code that computes the transpose of a matrix. The conditional `throw` statement (and the associated catch) is indicative of any statement that transfers the control (e.g., break, goto, return, or `throw` statement) from the body of the loop nest to an instruction outside the loop nest. The condition could be any of the possible sanity checks (e.g., array index bounds check) relevant in this context.

Figure 1b shows an incorrectly tiled loop [24], obtained by ignoring the control flow resulting from the conditional `throw` statement present in Figure 1a. The tiles are of size `bn*bn` and for simplicity, we assume that `M` and `N` are multiples of `bn`. It can be easily seen that the performed loop tiling is not semantics preserving (due to the control dependence). Similar reasoning can be given for many of the loop optimizations (e.g., software pipelining [24], loop interchange [24]), which makes them non-applicable, in the presence of exceptions.

Most of the compilers (e.g., GCC [32]) tend to be conservative and do not invoke many loop optimizations in the presence of exception throwing statements (inefficient). Some compilers (e.g., XLC [17]) provide command line switches to disable exceptions altogether and invoke the traditional optimizations (semantically non-preserving). There have also been many studies to identify unnecessary exception `throw` statements [8, 18, 7], and mark *exception-safe* regions [23]. While such a process can enable aggressive optimizations, it has its limitations owing to the specialized techniques used to target specific popular exceptions (for example, NullPointerException and ArrayIndexOutOfBoundsException) and the extent of information available at the time of compilation. To address these issues, we present a scheme to do *exception-safe* loop optimizations, especially when the optimizations may reorder the loop iterations. We define a loop optimization to be *exception-safe*, if it can be applied even on exception throwing loops, in a semantics preserving manner.

Though, in this paper for pedagogy, we use exceptions (as the control flow mechanism) and C++ (as the language of illustration), the presented concepts are equally applicable in the presence of other control flow statements (such as, goto, break, return) and programs written in a variety of high level languages that benefit from loop tiling.

Our Contributions

- A general scheme of backup, rollback (of the *advanced* iterations – ones that the unoptimized code would not have executed, if an exception is thrown) and safe-execution (of the *delayed* iterations – ones that the unoptimized code would have executed, but not executed by the optimized code) that can be used to derive the exception-safe variation of any existing loop optimization.

- Considering the importance of loop-tiling [24, 35], we specialize the exception-safe loop optimization scheme to derive an optimized exception-safe loop tiling scheme (oESLT – backs up minimal number of elements, and in case an exception is thrown, rolls back only the updates of the advanced iterations, and safely executes the delayed iterations). We also present a naive exception-safe loop tiling scheme (nESLT) that backs up every element and does a full rollback and safe-execution, in case an exception is thrown.
- We present a new program region called Extended SCoP (ESCoP), to help identify loops (with exception exits) that can be tiled.
- We implemented nESLT and oESLT in the Graphite framework of GCC 4.8. We present an evaluation over a set of base kernel benchmarks drawn from the popular PolyBench 3.2 benchmark suite [27]. We show that in the common case, when no exceptions are thrown, oESLT leads to significant performance gains (geometric mean 41.5%) compared to the base kernels (compiled using gcc -O3), at the cost of a minor memory overhead (geometric mean 0.3%). In this case, nESLT leads to similar gains in performance (geometric mean 40.8%), but incurs a much larger memory overhead (geometric mean 100%). If an exception is thrown, the impacts of oESLT and nESLT vary depending on the iteration in which the exception is thrown.

1.1 Related Work

Loop optimizations [24, 35] are arguably the most important optimizations implemented in the modern day compilers. Most of the recent advancements [2, 28, 29] in this space focus on improving the efficiency of the existing techniques. However, these works have mostly targeted programs that do not throw exceptions. The work of Yun et al [36] presents an optimal software pipelining scheme in the presence of simple control flow statements. In their scheme, software pipelining can be performed on code that includes control flows within the loop body. However, it still cannot handle loops that include exceptions or any other arbitrary jump statements that transfer the control out of the loop nest. Recently there has been increased interest [26] in designing optimizations for task parallel programs that may throw exceptions. Benabderahmane et al. [5] claim that the restrictions imposed by the polyhedral model are largely artificial and propose changes to the whole polyhedral optimization process such that the model can be more widely applied to the whole functions in a program. But it is not clear how their proposed scheme works in the presence of exceptions. To the best of our knowledge, ours is the first paper on exception-safety of traditional loop optimizations in general and exception-safe loop tiling in particular.

Analysis of programs that may throw exceptions has received a fair amount of interest, owing to the inherent scalability related issues therein. Sinha and Harrold [30] describe the effect of exception-handling constructs on the traditional control and data flow analyses and presents techniques to construct efficient representation for programs that may throw exceptions. Allen and Horwitz [1] use a similar representation to compute accurate slices, for programs that may throw exceptions. Choi et al [9] propose a compact representation of Control Flow Graph, called Factored Control Flow Graph (FCFG), for the exception related control flow in OO languages. Fu and Ryder [13] describe a static analysis technique that computes chains of semantically-related exception-flow links that can help improve the precision of control flow analysis. In contrast to these program analysis techniques, we present a technique to do exception-safe loop tiling, and the proposed analysis and transformation technique is not constrained by any scalability related issues.

Converting parts of programs that may throw exceptions to code that may not (e.g., by eliminating array out of bounds checks, null pointer checks and so on) is one of the

popular ways [7, 34, 8] to optimize programs that may throw exceptions. Loop versioning [22] is another promising approach where the compiler generates a specialized code with no exception `throw` statements; this specialized code is predicated with a series of checks that guarantee that no exception will be thrown. The main issue with these approaches is that they are specific to the pre-decided exceptions under consideration and it is quite challenging to extend the same to arbitrary conditional exceptions. In contrast, our proposed approach can handle any type of conditional `throw` statements, and we propose an extension to the base loop tiling optimization in the presence of exceptions.

Gupta et al [14] present a scheme to speculatively optimize the code, assuming that exceptions are never thrown. If an exception is thrown in the optimized code, they execute the unoptimized code (called the compensation code); this requires the complete backup of the updated array. In contrast, oESLT takes advantage of the underlying behavior of the loop tiling optimization to reduce the amount of backup, rollback and safe-execution.

There has been prior work on improving the scope and efficiency of speculative execution [4, 21, 10]. The main efficiency consideration here is that of minimal overhead and efficient recovery code generation. Another form of speculative execution with recovery is seen in the context of transactional memory [16, 12]. Our proposed method bears some resemblance to speculative execution, wherein we execute many tiles of the loop in a speculative manner and if an exception is thrown, we perform 'recovery'.

Song and Li [31] propose a scheme to tile loops speculatively, where the loop body may terminate prematurely because of convergence tests. In contrast to our proposed oESLT scheme, their scheme backs up all the array elements a priori (we only do partial backup), they backup the whole array many times, and their rollback requires writing of the whole array from the backed up store.

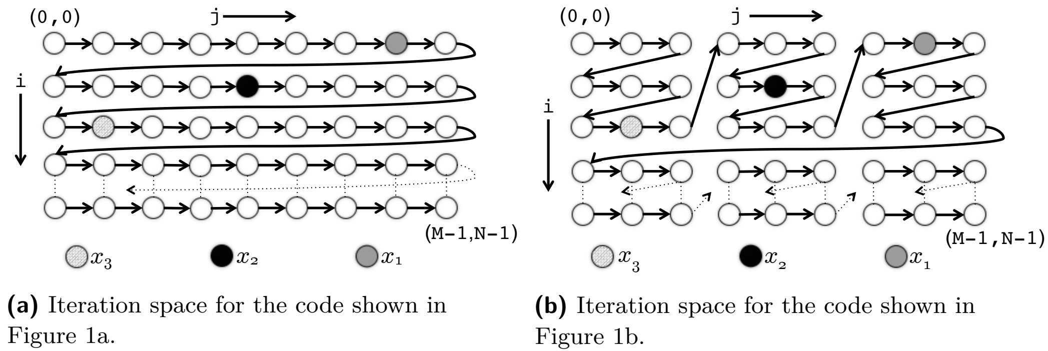
Our idea of backup and rollback has similarities to the work of versioning exceptions [25] that describes a language level extension, wherein the program can request a version of the store to be stored at a point of installation of a handler; when an exception is thrown, all the changes are reverted. Unlike our work, where for efficiency consideration the rollback is partial, the rollback in case of versioning exceptions is complete. It would be interesting to extend their work to handle arrays and partial rollback, and then using versioning exception to automatically generate rollback and safe-execution code, for loop tiling.

Outline: The rest of the paper is organized as follows. Section 2 presents a general scheme of exception-safe loop optimizations. Section 3 explains the general process of performing exception-safe loop tiling, along with a naive version thereof. Section 4 explains the scheme of optimized exception-safe loop tiling. We present the implementation and experimental results in Section 5 and conclude in Section 6.

2 Loop Optimizations in the Presence of Exceptions

In this section, we present techniques to do *exception-safe* loop-optimizations. Consider a normal loop nest L (consisting of loops with index variables $k_1, k_2, \ldots, k_n$) and a statement S present therein. Say, the free variables in S depend on a subset K of these index variables. Given a function $M : K \rightarrow Int$, we use $S(M)$ to represent the execution instances of S, where the index variable k_i gets the value $M(k_i)$, $\forall k_i \in K$. We now present some important concepts that form the basis of our exception-safe loop optimization scheme.

▶ **Definition 1.** Say the trace of the input loop consists of the sequence of statement instances $S(M_1), S(M_2), \cdots, S(M_n)$. If the same sequence forms the trace of the output loop, the transformation is said to be trace-preserving or else it is called trace-reordering.



(a) Iteration space for the code shown in Figure 1a.

(b) Iteration space for the code shown in Figure 1b.

Figure 2 Iteration space for the codes shown in Figure 1.

A trace-reordering transformation may result either via explicit reordering of the statements (example transformations: software-pipeline, instruction scheduling), or by reordering the iterations of a loop (example transformations: loop interchange, and loop tiling). Similarly, example trace-preserving transformations include loop peeling, loop unswitching and loop unrolling. The effect of exceptions on these two classes of loop transformations varies.

Trace-preserving transformations: Since the transformations do not change the sequence of instructions executed at runtime, such transformations can be oblivious to the presence of exception throwing instructions in the input program.

Trace-reordering transformations: The effect of exceptions on trace-reordering transformations is more involved. We will illustrate the same using the example.

Figure 2a shows the iteration space traversal for the code shown in Figure 1a. Each cell (`i,j`) represents an iteration. Consider the three iterations labeled x_1, x_2 and x_3. For the input program in Figure 1a, the order of execution of these iterations is $x_1 \prec x_2 \prec x_3$. The operator $\prec$ enforces an executes before relation.

Consider Figure 2b that shows the iteration space traversal for the code shown in Figure 1b (assuming, `bn`=3). The execution order for the previously considered iterations x_1, x_2, and x_3, in the transformed code is $x_3 \prec x_2 \prec x_1$. That is, compared to the iteration x_2, the iteration x_3 has been *advanced* and the iteration x_1 has been *delayed*. This difference in the execution order comes into limelight, when `cond` evaluates to `true` (say at iteration x_2). In Figure 2a, for an exception to be thrown in iteration x_2, the iteration x_1 must have been executed, and the iteration x_3 would not be executed after the exception is thrown. However, for the iteration space traversal shown in Figure 2b, for an exception to be thrown in iteration x_2, the iteration x_3 would have been executed, and the iteration x_1 would not be executed after the exception is thrown. Thus, the semantics of the transformed loop (Figure 1b) does not match that of the input loop (Figure 1a).

To make the code in Figure 1b *exception-safe*, if an exception is about to be thrown during the execution of the transformed loop (for example, at iteration x_2 in Figure 2b), the following additional steps need to be performed.

1. The execution of iterations that have been advanced (for example, iteration x_3) should be rolled-back (first rollback phase).
2. The delayed iterations (for example, x_1) should be executed in an order matching the traversal space shown in Figure 2a (safe-execution phase). If an exception is about to be thrown during any of these iterations (say, x_1) then roll-back the execution of all the iterations that have been advanced, with respect to x_1 (second rollback phase).
3. Throw the most recent exception object (from iteration x_1 or x_2).

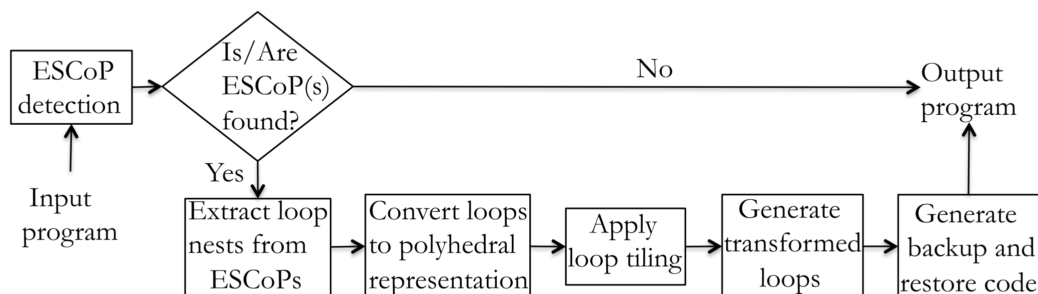


Figure 3 Exception safe loop tiling process.

These steps are applicable for any trace-reordering loop transformation. In this paper, we focus on loop tiling which is an important and popular trace-reordering transformation.

3 Exception safe loop tiling

Figure 3 shows the block diagram for our optimized exception safe loop tiling. We define a new type of program regions called Extended Static Control Parts (ESCoPs) (an extension of Static Control Parts (SCoPs) [5, 11]) that admit loops with possible exception related edges in the CFG. We then extract the loop nests from the ESCoPs and mark the control paths introduced by `throw` statements (*exception edges*). Next, we perform traditional loop tiling (blocking) on the extracted loop nests, which requires that there are no loop carried dependences (control or data) in the loop. Note: the loop carried control dependence introduced by the `throw` statements are masked by the ESCoPs. The tiled loop (for example, the code shown in Figure 1b) is then made *exception-safe* by emitting additional code to backup, to rollback and do safe-execution, if the input loop has exception edges.

Our proposed transformations handle only those loops which have affine bounds and affine conditions, but may contain `throw` statements. The conditions that bound the exception `throw` statements need not be affine as they are not modeled using the polyhedral representation. For the ease of explanation, we explain our techniques and algorithms over square tiling. However, the techniques are general enough to be extended to other shapes of tiles, provided the tiles are disjoint. We now explain the details of ESCoPs over which we implement our proposed algorithms and follow it up with a naive scheme of doing exception-safe loop tiling.

3.1 Extended SCoPs : Single Entry Multiple Exit Regions

A Static Control Part (SCoP) is defined as a region of consecutive statements comprising of loops with affine bounds and affine conditions, where the conditions depend only on either constants or loop invariant variables, or the loop index variables [5, 11]. The access functions used in a memory reference statement should be affine. The loops must be in canonical form.

An important feature of a SCoP is that all the loops that are a part of a SCoP have at most a single exit. Thus, a loop nest with multiple exits (arising due to a conditional `throw` statement present in the body of the inner most loop) is not part of a SCoP. Note, we need to ensure that the presence of exception edges do not inhibit the tiling of the input loop. To address these challenges, we define Extended SCoPs (ESCoPs) that admit SCoPs with a relaxation that the code therein may throw exceptions. The proposed ESCoPs have the following characteristics:

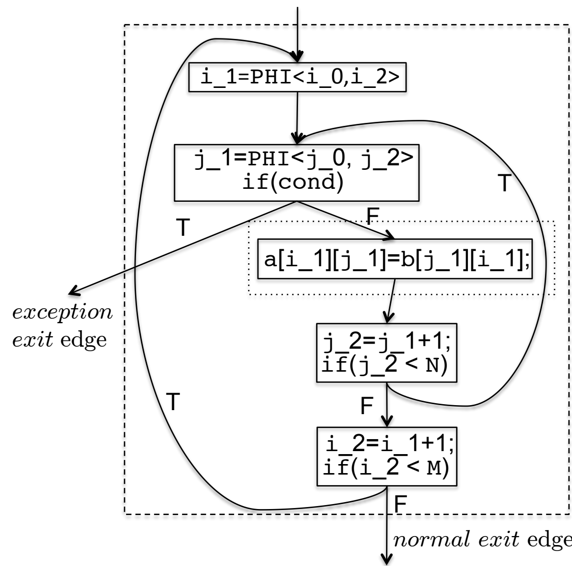


Figure 4 ESCoP and SCoP for the example code shown in Figure 1a.

1. ESCoPs are Single Entry Multiple Exit (SEME) regions.
2. Erasing all the throw statements from an ESCoP results in a SCoP – the underlying SCoP of the ESCoP.
3. The exit condition of the loop that is part of the underlying SCoP, of a given ESCoP, is called the *normal-exit condition* and the associated control flow edge is called the normal-exit edge. An ESCoP can have at most one normal-exit edge.
4. The exit conditions other than the normal-exit condition are termed *exception-exit* conditions (or *abnormal-exits* in GCC parlance), and the associated control flow edges are called the exception-exit edges.
5. Every loop belonging to an ESCoP need not have exception-exits. However, every loop belonging to an ESCoP should contain at most one normal-exit. Therefore, every SCoP is an ESCoP with zero exception-exits.

Figure 4 shows part of the control flow graph (CFG) for the loop nest shown in Figure 1a. The dotted region depicts the maximal SCoP detected, using the traditional SCoP detection algorithm [20], which unfortunately contains no loop. However, the corresponding ESCoP (depicted by a dashed box in Figure 4) includes the whole loop, along with the exception exit edge (the "T" labeled edge out of the second node). The "F" labeled edge out of the last node represents the normal-exit edge.

Given a program, we build its corresponding set of ESCoPs, such that each maximal loop nest present within a try-catch block or procedure boundary is associated with a unique ESCoP. Later, we perform exception-safe loop tiling on these ESCoPs. Our choice of ESCoPs is inspired from the fact that rollback and safe-execution operations have to be performed before the control is transferred to the exception handler. Thus, the rollback and safe-execution code can be inserted on the exception exit edges of the ESCoPs.

Note that in general, an ESCoP may contain multiple loops and some additional statements before/after the loops. For the ease of discussion, in this manuscript, we assume that each ESCoP has a single loop nest, which does not in anyway impact the generality of our proposed techniques.

```
for(i = 0; i < M; i++){ // backup phase
 for(j = 0; j < N; j++){
  bak[i][j] = a[i][j];
 } /* j */
} /* i */
try{
 try{ .. The loop nest of Figure 1b without the try-catch block ..
 }catch(...){ // all exceptions caught
   for(ti = 0; ti < (ii+1)*bn; ti++){ // rollback phase
    for(tj = 0; tj < N; tj++){
     a[ti][tj] = bak[ti][tj];
    } /* tj */
   } /* ti */

   for(i = 0; i < M; i++){ // safe-execution phase
    for(j = 0; j < N; j++){
     if(cond){ throw e; }
     a[i][j] = b[j][i]; } /* j */
   } /* i */
 } /* catch */
}catch(Ex e){ .. }
```

Figure 5 Impact of nESLT on the code shown in Figure 1a.

3.2 Naive Exception Safe Loop Tiling

The naive exception safe loop tiling (nESLT) approach speculatively tiles the loop assuming that no exceptions are thrown. It emits code (before the tiled loop) to back up all the elements that may be updated in the input loop (the backup phase). If an exception is thrown during the execution of the tiled loop then nESLT handles it in two phases. i) rollback: rolls back a set of elements that form an over-approximation of the actual updated elements, and ii) safe-execution: executes the untiled loop (from the beginning) till an exception is thrown. Note: if $\overrightarrow{I}$ and $\overrightarrow{J}$ are the iteration vectors when the exception is thrown in the tiled and untiled loop, respectively, then $\overrightarrow{I} \not\prec \overrightarrow{J}$ [24]. Further, if no exception is thrown, then the rollback and safe-execution phases are not invoked.

For the code shown in Figure 1a, Figure 5 shows the code generated by nESLT. The backup phase backs up all the elements of the array `a` into an array `bak`, and is used in the rollback phase, if an exception is thrown.

The main drawbacks of nESLT are that it conservatively does backup (all the elements, irrespective of when the exception is thrown), and rollback (all the updated elements and may be a few more) and safe-execution (more iterations than required). This can lead to a significant execution time and space overhead.

4 Optimized exception safe loop tiling

We now present our scheme of optimized exception-safe loop tiling (oESLT). We first explain some key ideas regarding backup, rollback, and safe-execution and then present the individual algorithms.

```
for(i=0;i<M;i++){
 for(j=0;j<N;j++){
  for(k=0;k<P;k++){
    if (cond) throw Ex;
    a[i][j]+= b[i][k]*c[k][j]; } } }
```

(a) complete-update-pattern

```
for(i=0;i<M;i++){
 for(j=0;j<N;j++){
   x[j] += b[i][j];
 }
}
```

(b) partial-update-pattern

Figure 6 Array update patterns.

4.1 Backup

Speculative execution of tiled loops may require certain updates to be rolled back and this in turn requires that the relevant older values are backed up. The exact values to back up depends on the array update pattern: *complete-update-pattern* (*cu-pattern*) or *partial-update-pattern* (*pu-pattern*).

cu-pattern: Consider the example code snippet shown in Figure 1a, where updates to a new element (for example, `a[i][j]`) start only after all the updates to the older elements (for example, `a[i][j-1]`) are complete. Such updates are termed as *complete-updates*. Figure 6a shows another example of code performing complete-update. At any iteration in this loop, there is at most one array element (the current element `a[i][j]`) that is partially updated. In case of complete-updates, all the distinct array elements updated in the previous iterations of the loop would never be updated again.

pu-pattern: In Figure 6b, each array element (for example, `x[j]`) is updated partially in each iteration of the outer loop. Such updates are termed as *partial-updates*. At any iteration in the loop, there are up to `N` array elements that are partially updated. If the updates to an array in a loop nest follow both cu-pattern and pu-pattern, we assume the array to be updated in the latter pattern only.

The goal of our optimized exception-safe loop tiling is to backup as few elements as possible and as few number of times as possible. For the sake of efficiency, we propose two different backup schemes: (i) Backup each element of the updated array, every time it is updated (backup for each iteration) – used for the array updates in input loop nest that follow the pu-pattern. (ii) Backup each element of the updated array exactly once (backup for each element) – used for the array updates in input loop nest that follow the cu-pattern.

4.1.1 Backup location and size

The location of insertion of the backup code and the size of the backup array also depend on the type of the array update pattern.

Case *pu-pattern*: Since we will use the first backup scheme (backup for each iteration), we backup just before every update and add the backup statements (copy) the innermost loop. Thus, the size of the backup array is bound by the size of the iteration space of the loop nest (e.g., the tiled version of the code shown in Figure 6b requires `M*N` space for backup).

Case *cu-pattern*: Even though the size of the updated array gives an upper limit on the size of the backup array, depending on the program point where the backup code is emitted (*backup point*), the actual size can be less. Interestingly, the backup point depends on the order in which the array elements (requiring backup) in the transformed loop are updated. The goal in this process (*efficiency consideration*) is to reduce the size of the backup array (reduces space overhead) and the number of elements backed up at one go,

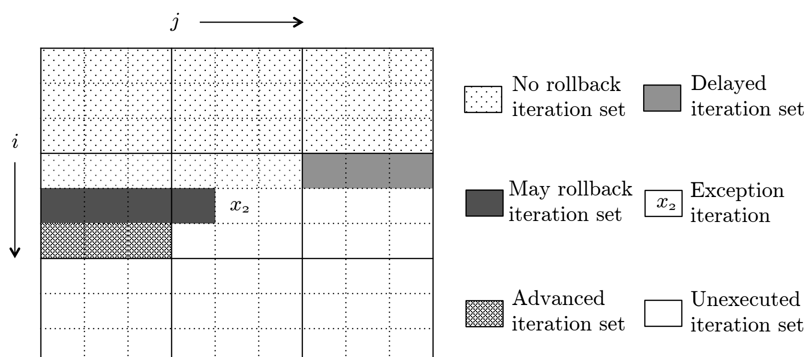


Figure 7 Advanced and delayed iterations.

```
for(ii=0;ii<M/bn;ii++){
 for(jj=0;jj<N/bn;jj++){
   /* Backup point. */

  for(kk=0;kk<P/bn;kk++){
   for(i=ii*bn;i<(ii+1)*bn;i++){
    for(j=jj*bn;j<(jj+1)*bn;j++){
     for(k=kk*bn;k<(kk+1)*bn;k++){
        if (cond) throw Ex;
        a[i][j]+=b[i][k]*c[k][j]; }}}
```

(a)

```
for(ii=0; ii<M/bn; ii++){
     /* Backup point */

 for(kk=0; kk<P/bn; kk++){
  for(i=ii*bn;i<(ii+1)*bn;i++){
   for(k=kk*bn;k<(kk+1)*bn;k++){
    for(jj=0; jj<N/bn; jj++){
     for(j=jj*bn;j<(jj+1)*bn;j++){
        if (cond) throw Ex;
        a[i][j]+=b[i][k]*c[k][j]; }}}
```

(b)

Figure 8 Two of the many possible tilings for the code shown in Figure 6a. For simplicity assume: `M`, `N` and `P` are multiples of `bn`. Backup code insertion point depends on the tiled code.

henceforth referred as *backup pulse* (reduces the execution time overhead, if an exception is thrown). We illustrate it with some examples.

To illustrate the possibility that the size of the backup array can be less than the size of the updated array, consider the tiled iteration space shown in Figure 7. We observe that for the first iteration of any row of tiles, the advanced iteration set and the delayed iteration set are empty. In other words, at the beginning of every row of tiles, the tiled loop execution is semantically equivalent to the unoptimized loop execution. Thus the advanced iteration set for any iteration contains the iterations only from the beginning of the current row of tiles and it is enough to backup these updates, and rollback in case an exception is thrown. This leads to a scenario where it is enough to allocate memory of size equaling that of a row of tiles, for the backup array. We now illustrate the importance of backup point, on the exact size of backup array and the backup pulse, with examples.

For the code shown in Figure 6a, Figure 8 shows two different tilings, and Figure 9 presents the backup codes thereof. Choosing any other program point in Figure 8 to emit the backup code would lead to increased overheads (space and execution time). For Figure 8a, inserting the backup code (within the loop with index `jj`) meets our specified efficiency consideration in the best possible manner (backup size = `bn*N`, backup pulse = `bn*bn`). Note that, it is enough to keep a backup of just one row of tiles at a time and for efficiency we can backup one tile at a time. Similarly, for Figure 8b, inserting the backup code (within

```
for(i=ii*bn;i<(ii+1)*bn;i++)
 for(j=jj*bn;j<(jj+1)*bn;j++)
  bak[i%bn][j]=a[i][j];
```

(a) Backup code for Figure 8a

```
for(i=ii*bn;i<(ii+1)*bn;i++)
  for(j=0;j<N;j++)
   bak[i%bn][j]=a[i][j];
```

(b) Backup code for Figure 8b

Figure 9 Backup codes.

the outermost loop with index `ii`) meets our specified efficiency consideration in the best possible manner (backup size = `bn*N`, backup pulse = `bn*N`). As another example, a tiling scheme that further exchanges the `ii` and `kk` loops of Figure 8b would lead to a scenario, where we have to backup the full array (size = `M*N`) all in one go (before the beginning of `kk` loop) – similar to nESLT.

4.1.2 Backup Algorithm

The backup algorithm is shown in Figure 10. The input loop nest *inpL* (by ignoring the exception-exit edges) is transformed into *trL* using the traditional loop tiling technique. Every array update statement in the *trL* is handled separately and hence a unique backup array is used for every array that is updated in the loop.

For the input array update statement S, say R is its counterpart in *trL*. If the array update of S in *inpL* follows the pu-pattern, the array element modified in S is backed up before each update. We use an auxiliary function 'emit' to generate code and || indicates a string concatenation operator. Note that the backup array, in this case, is indexed with the iteration vector of *trL*.

If the array update of S in *inpL* follows the cu-pattern, the backup process is more involved. First the backup-point is determined and then the backup loop nest is generated. The variables *inpIdxSeq* and *tiledGrpIndexSeq* contain an ordered set of loop index variables. *inpIdxSeq* contains the list of the loop-indices of *inpL*, corresponding to the loops that surround S; the index of the outermost loop comes first. For the code shown in Figure 6a, *inpIdxSeq* = [`i, j, k`]. During the first phase of traditional tiling transformation, the loop nest is strip-mined [24]. For each loop in the input loop, the strip-mined loop has two loops: one loop (the outer one) iterates over groups of elements (*grouping loop*), and another loop (the inner one) iterates over all the elements in a given group (*element loop*). The variable *tiledGrpIndexSeq* contains the list of loop indices of the grouping loops that surround R (for the code shown in Figure 8b, *tiledGrpIndexSeq* = [`ii, kk, jj`]). We use two auxiliary functions getIndexSeq and getGroupIndexSeq to return the appropriate ordered sets. Given a loop index of a loop, the auxiliary function OriginalIndex returns the loop index of the input loop (*inpL*). For example, OriginalIndex function can be represented as a set of pairs {(`ii, i`), (`i, i`), (`jj, j`), (`j, j`), (`kk, k`), (`k, k`)}.

We emit the backup code just before that outermost grouping loop, at which the sequence *inpIdxSeq* and OriginalIndex(*tiledGrpIndexSeq*) do not match. This is the point after which the update pattern of the input loop differs from the transformed loop. The backup code is determined by the chop [19] computed for the update statement R, bound by the loop over the index variable tiledGrpIndexSeq[d]. For the two possible tilings shown in Figure 8a and Figure 8b, Figure 9 shows backup code to be emitted at the backup point specified. Note: Inserting the backup code on the critical path of execution may lead to i-cache and d-cache pollution. Therefore it is desirable to have a backup mechanism in which the backup execution does not come on the critical execution path of the input code. Hence in case of

```
1  Function GenBackupCode (inpL, trL, S, bak)
   Input: inpL: input loop nest that has been tiled; trL: transformed tiled loop nest; S:
          array update statement in inpL; bak the backup array.
2  begin
3      Say arr is the array variable updated in S and R is the counterpart of S in trL;
4      Say →I =index vector used to update the array in R;
5      updateType = type of array update in S;
6      if (updateType = cu-pattern) then
7          inpIdxSeq = getIndexSeq(inpL, S);
8          tiledGrpIndexSeq = getGroupIndexSeq(trL, R);
9          for (d = 0 ; ; d++) do // Compute the longest common prefix length.
10             if inpIdxSeq[d] ≠ OriginalIndex(tiledGrpIndexSeq[d]) then break;
11             ;
12         bakSlice = chop(R, getLoop(tiledGrpIndexSeq[d]));
13         emit bakSlice before the loop with loop index tiledGrpIndexSeq[d];
14         emit bak || "[" || →I || "] =" || arr || "[" || →I || "]" as the last statement in the
           body of the innermost loop of bakSlice;
15     else
16         Say →I1 =iteration vector in which R is updated;
17         emit bak || "[" || →I1 || "] =" || arr ||"[" || →I ||"]" before R;
```

Figure 10 Backup Algorithm.

"Backup for each element", the backup code is not inserted inside the innermost loop (though it is semantically correct to do so).

4.2 Rollback and Safe-execution

In this subsection, we present techniques to generate an efficient code to rollback and perform safe-execution. The main intuition behind this process is to rollback only the relevant computations (advanced iterations) and perform safe-execution of just the required iterations (delayed iterations).

4.2.1 Computing advanced and delayed iteration sets

Given an unoptimized loop, its tiled counterpart, and an iteration vector $\overrightarrow{p}$, we identify two sets of iterations:

1. $A(\overrightarrow{p})$: The set of iterations executed before $\overrightarrow{p}$ in the tiled loop.
2. $B(\overrightarrow{p})$: The set of iterations executed before $\overrightarrow{p}$ in the unoptimized loop.

If an exception is thrown in the iteration vector $\overrightarrow{p}$, then the set $(A(\overrightarrow{p}) - B(\overrightarrow{p}))$ consists of all the advanced iterations (with respect to $\overrightarrow{p}$) and hence the updates made by only these iterations need to be rolled-back. Similarly, the set $(B(\overrightarrow{p}) - A(\overrightarrow{p}))$ consists of all the delayed iterations and hence all these iterations have to be freshly executed (as part of safe-execution, may be in an untiled manner). As discussed earlier in this section, if an exception is thrown during this phase of safe-execution, say at iteration $\overrightarrow{q}$, then we have to further roll-back the execution of all the iterations that have been advanced with

respect to $\overrightarrow{q}$ (second rollback phase). The set of advanced iterations with respect to $\overrightarrow{q}$, after performing roll back for the iteration $\overrightarrow{p}$ is $((A(\overrightarrow{q}) - B(\overrightarrow{q})) - (A(\overrightarrow{p}) - B(\overrightarrow{p})))$. Note that $((B(\overrightarrow{q}) - A(\overrightarrow{q})) - (B(\overrightarrow{p}) - A(\overrightarrow{p}))) = \phi$, and hence there are no delayed iterations with respect to $\overrightarrow{q}$. As a result no further exceptions may be thrown and we do not need any further rollback phase. Note that if the unoptimized execution of the loop throws an exception, then the iteration at which the exception is thrown is given by $\overrightarrow{q}$ or $\overrightarrow{p}$, depending on whether any exception is thrown during safe-execution or not, respectively.

As an example, consider Figure 7 that shows the iteration space for the matrix transpose examples shown in Figure 1 (assuming `M=N=9` and a tile size of 3×3). Consider a specific iteration x_2. The set $A(\overrightarrow{x_2})$ is given by the iterations corresponding to the dotted, dark-grey and checked boxes. The set $B(\overrightarrow{x_2})$ is given by the iterations corresponding to the dotted, dark-grey and the light-grey boxes. Thus, if an exception is thrown at x_2, the checked-boxes $(= A(\overrightarrow{x_2}) - B(\overrightarrow{x_2}))$ correspond to the iterations that need to be rolled-back, and the iterations corresponding to the light-grey boxes $(= B(\overrightarrow{x_2}) - A(\overrightarrow{x_2}))$ should be safely executed.

For the loop in Figure 6a and its corresponding transformed code in Figure 8b (along with the backup code in Figure 9b), the loop with rollback code and safe-execute code is shown in Figure 11. For brevity we show the rollback and safe-execution loops for two of the total five interchanges (algorithms given below).

4.2.2 Generating efficient rollback and safe-execution code

An interesting challenge that was observed doing the generation of the above mentioned $A(\overrightarrow{p})$ and $B(\overrightarrow{p})$ sets is that these sets are non-convex in general. Performing the set-difference operations (in GCC) over non-convex sets to generate the sets corresponding to the advanced and delayed iterations, and the automatic generation of loop nests over such sets was (i) quite time consuming, and (ii) leading to inefficient code (owing to the complex convex set decomposition routines employed by the underlying library code). To overcome these challenges, we first identify the impact of loop tiling (in terms of reordered operations) as the union of the impact due to its constituent sub-transformations, and then generate the rollback and safe-execution code corresponding to each of these sub-transformations.

The tiling of a loop nest can be seen as strip-mining followed by a series of loop interchanges. The strip-mining transformation does not reorder the iteration space and only the loop interchange operations contribute to the final reordering. For the tilings shown in Figure 8a and Figure 8b, the sequence of interchanges are [(`j`, `kk`), (`i`, `jj`), (`i`, `kk`)] and [(`j`, `kk`), (`j`, `k`), (`jj`, `kk`), (`jj`, `k`), (`i`, `kk`)], respectively. An interchange (`p`, `q`) indicates that the outer-loop with index variable `p`, is interchanged with the inner-loop with index variable `q`.

For an iteration vector $\overrightarrow{p}$ over some iteration space, the advanced iterations set $A(\overrightarrow{p}) - B(\overrightarrow{p})$ and delayed iterations set $B(\overrightarrow{p}) - A(\overrightarrow{p})$ produced by a single interchange operation are convex sets. However, for an iteration vector $\overrightarrow{q} \in B(\overrightarrow{p}) - A(\overrightarrow{p})$ ($\overrightarrow{q}$ is an iteration of safe-execution phase in which an exception is thrown), then $((A(\overrightarrow{q}) - B(\overrightarrow{q})) - (A(\overrightarrow{p}) - B(\overrightarrow{p})))$ may not be a convex set. In case of the "Backup for each element" scheme, for simplicity, without loss of significant performance, we over-approximate this set to be the minimal enclosing convex set possible (max overhead bound by the tile size). In case of the "Backup for each iteration" scheme, we decompose the non-convex set into a series of convex sets.

4.2.3 Rollback and Safe-execution algorithm

Figure 12 shows the driver for generating the rollback and safe-execution code. The function InvokeRestorePhaseCode takes as input an auxiliary structure *trLStruct* that contains all

```
try{
  .. Code from Figure 8b, along with Figure 9b ..
}catch(...){ /* Rollback code start */
 for(p=i+1;p<(ii+1)*bn;p++){// Rollback for (i, kk) plane
  for(q=0;q<N/bn;q++){
   for(r=q*bn;r<min(N,(q+1)*bn);r++){
    a[p][r]=bak[p%bn][r]; } } }
 p=i; // Rollback for (jj, kk) plane
 for(q=jj+1;q<N/bn;q++){
  for(r=q*bn;r<min(N,(q+1)*bn);r++) {
   a[p][r]=bak[p%bn][r]; } }
               ··· Rollback for other planes. Not shown ...

 try{  /* Safe Execution code start */
  // Safe-execution for (i, kk) plane
  for(p=ii*bn;p<i;p++){
   for(q=0;q<N/bn;q++){
    for(r=q*bn;r<min(N,(q+1)*bn);r++){
     for(s=kk+1;s<P/bn;s++){
      for(t=s*bn;t<min(P,(s+1)*bn);t++){
       if (cond) throw Ex;
       a[p][r]+= b[p][t]*c[t][r]; } } } } }
  p=i;  // Safe-execution for (jj, kk) plane
  for(q=0;q<jj;q++){
   for(r=q*bn;q<min(N,(q+1)*bn);r++){
    for(s=kk+1;s<P/bn;s++){
     for(t=s*bn;t<min(P,(s+1)*bn);t++){
      if (cond) throw Ex;
      a[p][r]+= b[p][t]*c[t][r]; } } } }
           ··· Safe execution code for other planes. Not shown ...

 }catch(...){ /* Second rollback code start */
  for(x=p+1;x<=i;x++){ // Rollback for (i, kk) plane
   for(y=0;y<N/bn;y++){
    for(z=y*bn;z<min(N,(y+1)*bn);z++) {
     a[x][z]=bak[x%bn][z]; } } }
  x=i; // Rollback for (jj, kk) plane
  for(y=q+1;y<=jj;y++){
   for(z=y*bn;z<min(N,(y+1)*bn);z++) {
    a[x][z]=bak[x%bn][z]; } }
          ··· Second rollback for other planes. Not shown ...

  throw; // throw the last caught exception (second rollback code)
 } // end of second rollback and safe execution
 throw; // throw the last caught exception (first rollback code)
} // end of first rollback code.
```

Figure 11 Restore code.

```
1 Function InvokeRestorePhaseCode (trLStruct)
  Input: trLStruct: contains all the required information about tiled loop nest
2 begin
3   for (every exception-exit edge e in the tiled loop nest of trLStruct) do
4     e1 = GenRollbackCode(e, trLStruct); // Rollback phase
5     GenSafeExecutionCode(e1, trLStruct); // Safe-execution phase
```

Figure 12 Driver for rollback and safe-execution phases.

```
1 Function boundedSubtract (LoopB, LoopA, T, →I)
  Input: LoopB is the LoopNest on which a loop interchange operation is performed to
         obtain the loop nest LoopA; →I is an iteration vector; T is a statement in the
         body of LoopB and LoopA
2 begin
3   A=getDomain(LoopA, T, →I);
4   B=getDomain(LoopB, T, →I);
5   return subtract(A,B);
```

Figure 13 Parameterized bounds computation, followed by set subtraction.

the information about the tiled loop nest. For each exception edge in the tiled loop nest, the function InvokeRestorePhaseCode invokes the rollback phase code generator GenRollbackCode and the safe-execution phase code generator GenSafeExecutionCode. The rollback code is inserted at the destination of the exception edge. The exit point of the rollback code (returned by GenRollbackCode) is used by GenSafeExecutionCode to insert the safe-execution code.

Figures 13 and 14 show the sketch of two helper algorithms. The function boundedSubtract (Figure 13) takes as input two loop nests (derived by interchanging two loops therein), along with a common statement, and an iteration vector. The function getDomain ($LoopA$, T, $\overrightarrow{I}$) returns the set of iterations of $LoopA$ (executed before $\overrightarrow{I}$) in which T is evaluated. The boundedSubtract function returns the set-difference of these domains.

The function getRestoreStmt (Figure 14) takes an array update statement R as input and returns a statement T (that acts as the restore statement for R). The domain of T is set to the vector space of the index vector of R, if we use the "backup per element" scheme. Otherwise, it is set to the vector space of the iteration vector of the loop containing R.

The algorithm for the rollback is shown in Figure 15. Rollback loops are emitted to restore the computations performed in the advanced iterations. They are emitted for each interchanged plane in *interchanges*. The function getInterchanges returns the sequence of interchange operations performed on the input loop nest that produced the tiled loop nest trL. Each such interchange has an associated input loop and output loop.

A rollback loop is generated for each interchange operation. For each update statement in the body of the loop nest, we compute the restore statement (by invoking getRestoreStmt). We compute the iterations to rollback, by invoking *boundedSubtract* on the domains of trS and trL, and store in *itrsToRollBack*. We then project this set onto the domain of T, to compute the actual domain in which the rollback should happen. All such generated restore statements are fed to a loop generator (generateLoops) to generate the loop nest for the

```
1  Function getRestoreStmt (R)
   Input: R : an array update statement that is backed up
2  Say arr[→I] is the array updated in R;
3  if (backupScheme = “backup per iteration”) then
4  |   Say T is the statement arr[→I] = bak[→I1], where →I1 is the iteration vector of the
   |   loop containing R;
5  |   domain(T)=vectorSpace(→I1);
6  else // backup per element
7  |   Say T is the statement arr[→I] = bak[→I];
8  |   domain(T)=vectorSpace(→I);
9  return T;
```

Figure 14 A helper function to generate a restore statement.

```
1  Function GenRollbackCode (e, trLStruct)
   Input: e : entry edge to place the generated code; trLStruct: contains all the
          required information about tiled loop nest;
2  begin
3  |   interchanges = getInterchanges(trLStruct);
4  |   while (interchanges.hasNext()) do
5  |   |   op = interchanges.next();
6  |   |   trS = op.inputLoop(); trL = op.outputLoop();
7  |   |   for (every array update statement R in trS) do
8  |   |   |   T = getRestoreStmt(R);
9  |   |   |   →p = iteration vector used to evaluate R in trS.
   |   |   |   itrsToRollBack=boundedSubtract(trS, trL, R, →p);
10 |   |   |   domain(T) = Project(itrsToRollBack, domain(T));
11 |   |   |   rollbackStmts.add(T);
12 |   |   ⟨exitEdge, newLStruct⟩ = generateLoops(e, rollbackStmts, trLStruct);
13 |   |   e = exitEdge ;
14 |   return exitEdge;
```

Figure 15 Rollback phase.

rollback phase. This function returns the new loop (*newLStruct*) and the normal exit point (*exitEdge*) of *newLStruct*. Figure 11 shows the rollback phase code for the two interchanges (`i`, `kk`) and (`jj`, `kk`).

The rollback loop generation phase is followed by the safe-execution loop generation phase. The safe-execution phase consists of two steps. In the first step, all the loop nests that execute the delayed iterations is generated. In the next step, a second set of rollback loops are generated which rollback the advanced iterations if an exception is thrown during the execution of the delayed iterations. We want to ensure that the delayed iterations are executed in the unoptimized order (so that if any exception is thrown during this execution, it matches the first exception that is thrown during the unoptimized execution). To aid in this process, we define a binary relation ‘$<_o$’ over a pair of interchange operations. Say $op_a = (a_1, a_2)$ and $op_b = (b_1, b_2)$, we say that $op_a <_o op_b$, if a_1 is outer to b_1 (covariant) in

```
1  Function GenSafeExecutionCode (e, trLStruct)
   Input: e : entry edge to place the generated code, trLStruct: contains all the
          required information about the generated tiled loop nest.
2  begin
3  |  interchanges = getInterchanges(trLStruct).sort('<_o');
4  |  emit "try {";
5  |  while (interchanges.hasNext()) do
6  |  |  op = interchanges.next();
7  |  |  trS = op.inputLoop(); trL = op.outputLoop();
8  |  |  for (every statement S in trL) do
9  |  |  |  p⃗ = iteration vector used to evaluate S in trL.
   |  |  |  Domain(S)=boundedSubtract(trL, trS, S, p⃗);
10 |  |  |_ safeExecStmts.add(S);
   |  |  // generate the loop at edge e.  exitEdge represents the
   |  |     normal-exit edge
11 |  |  ⟨exitEdge, newLStruct⟩ = generateLoops(e, safeExecStmts, trLStruct);
12 |  |  op.setSafeExecLoop(newLStruct);
13 |  |_ e = exitEdge;
14 |  emit "} catch (Exception ex) {";
15 |  interchanges = getInterchanges(trLStruct);
16 |  while (interchanges.hasNext()) do
17 |  |  op = interchanges.next();
18 |  |  trS = op.inputLoop(); trL = op.outputLoop();
19 |  |  nlS = op.safeExecLoop();
   |  |  // Second Rollback phase.  Will not throw any exception.
20 |  |  for (every array update statement R in trL) do
21 |  |  |  T = getRestoreStmt(R);
22 |  |  |  p⃗ = iteration vector used to evaluate R in trL;
23 |  |  |  q⃗ = iteration vector used to evaluate R in nlS;
24 |  |  |  P=boundedSubtract(trL, trS, R, p⃗); // A(p⃗) − B(p⃗)
25 |  |  |  Q=boundedSubtract(trL, trS, R, q⃗); // A(q⃗) − B(q⃗)
26 |  |  |  Domain(T)=subtract(Q,P); // (A(q⃗) − B(q⃗)) − (A(p⃗) − B(p⃗))
27 |  |  |_ add(restoreStmtList, T);
28 |  |_ g = generateLoops(g, restoreStmtList, trLStruct);
29 |_ emit "} // end catch";
```

Figure 16 Safe Execution phase.

the input strip-mined loop, or $a_1 = b_1$ and a_2 is inner to b_2 (contravariant) in the input strip-mined loop. Given two interchange operations I_1 and I_2, if $I_1 <_o I_2$, then the execution of the delayed iterations resulting from the interchange operation I_1 precedes that of I_2, in the unoptimized order.

Figure 16 shows the algorithm to generate the safe-execution phase code. To generate the safe-execution code, we process the ordered loop-interchanges (sorted using the comparator '$<_o$'). For every statement (includes array update statements and conditional exception exits) in *trL*, the safe-execution phase domain is computed similar to that of the rollback

phase. The generated statements are passed to the function generateLoops to generate a new loop (stored in *newLStruct*). To handle any exception that may be thrown during the safe-execution code, we insert the safe execution code inside a try-block and generate the code for the handler (second rollback phase, see Section 4.2.1) in the corresponding catch-block. Figure 11 shows the code generated by the algorithm shown in Figure 16.

We highlight some of the interesting points about the code generation task: (i) The code for the rollback and safe-execute phases is added at the destination of the exception edge. This code is executed only when the abnormal-exit edge is taken, thereby leaving the i-cache unpolluted during the execution of the main loop. (ii) We generate the safe-execution loops in an order that ensures that given two interchanges I_1 and I_2 (say, $I_1 <_o I_2$), if an exception may be thrown in both the safe-execution loops generated for I_1 and I_2, at iteration vectors $\overrightarrow{p_1}$ and $\overrightarrow{p_2}$, then it can be guaranteed that $p_1 \prec p_2$. This is in accordance of our guarantee that we don't need any further rollback phase (see Section 4.2).

4.3 Discussion

Bounds: To explain the bounds on the number of backup elements, restore operations and safe-execution operations (for oESLT), we use Figure 1a as an example. We will assume that the tile is of size `bn*bn`.

The minimum size of the backup array is at least `bn*N` and the maximum size of the backup array is bound by the size of the array that is being updated (`M*N`), and this scenario occurs when the selected backup-point is outside the loop nest (oESLT, behaves like nESLT).

The minimum number of elements that need to be restored is 0 (for example, if an exception is thrown in the very first iteration). And the maximum number of elements that may be restored is `(bn-1)*(N-bn)`; for example, this case occurs, when the exception is thrown at the iteration `i=bn, j=N-bn`.

The minimum number of safe-execution operations performed is 0 (for example, if an exception is thrown at the iteration `i=bn, j=0`). The maximum number of safe-execution operations performed is bound by the maximum number of restore operations: `(bn-1)*(N-bn)`.

Handling exceptions and segmentation faults in backup code: The backup code may throw an exception (for example, NullPointerException or ArrayIndexOutOfBoundsException in languages like Java) or may lead to segmentation fault (in languages like C++) if the array to be backed up is uninitialized or if the array access is illegal. This issue can be addressed by performing the required checks on the updated array, before the execution of exception-safe tiled code. For example, consider the tiled code shown in Figure 8a and its backup code shown in Figure 9a. The check (`a!=NULL && size_a[0]>=M && size_a[1]>=N`), where `size_a[0]` represents the size of outer dimension (first dimension) of array `a`, is performed before the execution of the tiled code. The tiled code is executed, only if the check succeeds. Otherwise, the unoptimized code is executed.

Importance of ESLT: Even though exceptions may be thrown rarely, semantics preserving compilation requires that we cannot apply traditional optimizations (such as loop tiling) by ignoring exceptions. Even though our proposed semantics preserving optimizations do incur some minor overheads, we show in Section 5 that the resulting gains are significant.

5 Implementation and Evaluation

We have implemented our proposed techniques (nESLT and oESLT) in the Graphite framework of GCC-4.8. The Graphite framework provides support for polyhedral optimizations

Sl	Kernel name	Input Size
1	2mm(2)	4000 (EL)
2	3mm (3)	4000 (EL)
3	gemm (1)	4000 (EL)
4	syrk (1)	4000 (EL)
5	syr2k (1)	4000 (EL)
6	doitgen (1)	256 (L)

Figure 17 Benchmarks kernels and the input sizes. The numbers in the brackets indicate the number of tiled loops. EL = Extra Large, L = Large.

like loop tiling, loop interchange and so on, and is implemented as an optimization pass in GCC-4.8 (one of the 100s of passes). We use the powerful Integer Set Library (ISL) [33] of GCC compiler framework to implement the helper functions (discussed in Section 4.2) like 'boundedSubtract', 'Project', 'subtract', 'getDomain', and support representations like domain of a statement and iteration vectors. The 'generateLoops' function used in Section 4.2, uses the underlying Chunky Loop Generator framework (CLOOG) [3], to generate the transformed loops.

We have evaluated our techniques on the popular Polybench 3.2 (converted to C++) benchmark suite [27]. It is a collection of well known numerical and linear-algebraic kernels (containing loops) designed specifically for the study of different loop optimizations. The kernels can be run over a wide set of input sizes. Compared to application benchmark suites like SPEC [15] and PARSEC [6], Polybench (because of its small size and focussed loop kernels) has an advantage that it helps us localize the impact of the specific loop optimization under consideration, with little interference from the rest of the program. To evaluate the different overheads associated with our techniques, we need benchmarks where exceptions may be thrown in different computation loops (or in the absence of such benchmarks, we have to edit the loops in the existing benchmarks to throw exceptions conditionally). Considering the size and complexity of the benchmarks like SPEC or PARSEC, it is quite challenging to introduce such tunable conditional `throw` statements in all the loops therein. In contrast, because of the smaller sizes, the Polybench kernels make it quite easy to introduce such tunable conditional `throw` statements.

We converted all the thirty Polybench kernels to C++. Of these kernels, we found that only the six kernels shown in Figure 17 could be tiled and the rest could not be tiled (fourteen of the kernels had data dependences that could not be resolved, and another ten were found to be non profitable by GCC). To derive the base (unoptimized) versions of the six shown kernels, we performed a pass of loop-distribution (to generate perfectly nested loops), and perform function inlining of the computation kernel in the 'main' method (to satisfy some requirements of the underlying GCC compiler framework to perform tiling). We then added a conditional `throw` statement before the main computation statement, and nested each such loopnest inside a try-catch block. Figure 18 shows the sample transformations done in the 2mm kernel, to derive the corresponding base kernel. We introduce the surrounding try-catch block, so that the code after the loop nest (for example, code to print the execution statistics) can execute, even if the exception is thrown. Note that the conditional exception `throw` statements are added in the innermost loop before the array access statement so as to give an effect of some common `throw` statements like ArrayBoundsCheck, NullPointerCheck, and so on. We use these base versions as the input to our loop tiling phase. We chose the largest input (provided by PolyBench) such that the arrays can be allocated on the stack (a

```
main(){
  ...
  mm();
  ...
  print (execTime);
}

void mm(){
 for (i=0;i<NI;++i){
  for (j=0;j<NJ;++j){
   C[i][j] = 0; /* init */
   for (k=0;k<NK;++k){
    C[i][j]+=alpha*A[i][k]*B[k][j];
   }
  }
}
```

(a) Representative snippet of the original Polybench kernel 2mm.

```
main(){
 ...
 for (i=0;i<NI;i++){
  for (j=0;j<NJ;j++){
   C[i][j] = 0; /* init */ } }
 try{
  for (i=0;i<NI;i++){
   for (j=0;j<NJ;j++){
    for (k=0;k<NK;++k) {
     if (some-cond)
      throw 20;// throw some exception
     C[i][j]+=alpha*A[i][k]*B[k][j];
  } } } } catch (...) {}
 ...
 print (execTime);
}
```

(b) Modified 2mm kernel.

Figure 18 Typical transformations done on the PolyBench kernels to derive the base (unoptimized) versions.

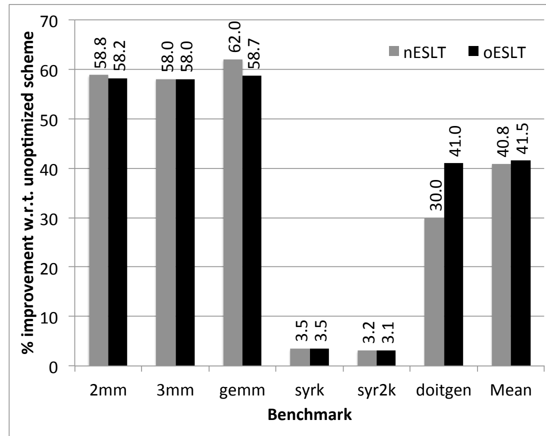
requirement of GCC loop-tiling). All the experiments were performed for square tiles. The size of the tiles was fixed by choosing the best tile size for the original benchmark kernels on our hardware: 16 for doitgen, and 20 for the rest.

All the experiments were performed on an Intel Xeon CPU E5-2670 system (with 32 KB L1 data and instruction caches, 256 KB L2 cache, and 20MB L3 cache). We present our evaluation over three different schemes: nESLT, oESLT, and the in built GCC loop tiling (here after termed as the unoptimized scheme). The unoptimized scheme does not tile any of these loops because of the presence of `throw` statements. Each of these schemes were invoked in GCC with -O3 -fgraphite options.

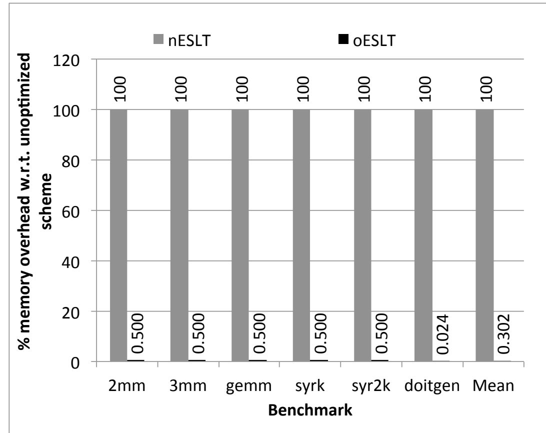
5.1 Impact of oESLT and nESLT when no exceptions are thrown

Figure 19 shows the impact of exception-safe loop tiling (ESLT) , when no exceptions are thrown inside the tiled loop; we measure the impact on both the execution time (Figure 19a) and memory (Figure 19b). We define percentage (%) improvement in execution time of scheme A over scheme B $= 100 \times (1 - \frac{\text{execution time with A}}{\text{execution time with B}})$. Similarly, we define percentage (%) overhead in the parameter under consideration (memory or execution time) of scheme A over scheme B $= 100 \times (\frac{\text{parameter value with A}}{\text{parameter value with B}} - 1)$. From Figure 19a, it can be seen that compared to the unoptimized scheme, on average oESLT and nESLT result in 41.5% and 40.8% improvements, respectively.

It can be seen that for most of the benchmarks (except doitgen) the impact of nESLT and oESLT on the execution time is similar; the maximum gap occurs for gemm, where the "backup all elements in one shot" scheme of nESLT helps in improved locality and leads to minor improvement. However, in case of doitgen, oESLT performs significantly better than nESLT. This is because, doitgen has a loop nest of depth 4, and during tiling, the outermost loop in the loop-nest is not tiled. Interestingly, the array update is based on the index of



(a) Percentage (%) improvement in execution time.



(b) Percentage (%) memory overhead.

Figure 19 Effect of ESLT when the nested conditional exception is never thrown.

the outermost three loop indices. As a result the number of elements in the backup pulse of oESLT is significantly small and hence the backup loop behaves as a prefetch loop, thereby improving performance.

Figure 19b shows the memory overhead incurred by both nESLT and oESLT, compared to the unoptimized scheme. It can be seen that the nESLT incurs significant memory overhead (geometric mean 100%). Compared to that oESLT reuses the backup space and reduces the memory overhead to a large extent (geometric mean 0.302%).

5.2 Overheads due to backup

To measure the overheads that we incur due to the insertion of backup code, we created a new ESLT scheme called the ideal ESLT (iESLT) scheme, wherein the loop is tiled, but has no backup code. Note: the conditional `throw` statements are retained (similar to oESLT and nESLT). In Figure 20, we compare the behavior oESLT and nESLT schemes against that of iESLT, when the input codes do not throw any exception. It can be seen that for most kernels the overhead due to the backup code is quite low (between -2% to 8.5%, geometric mean 1.39% for oESLT, and between 0% to 17%, geometric mean 2.69% for nESLT). In case

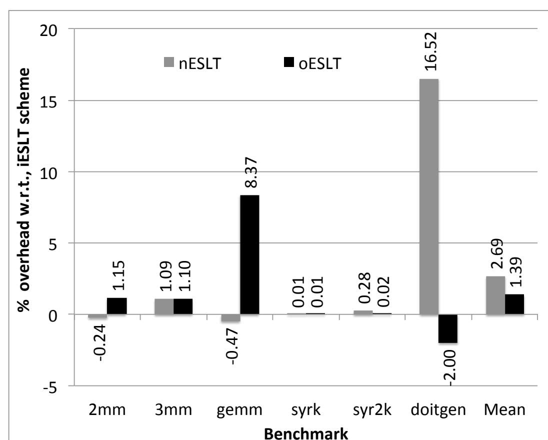


Figure 20 Percentage (%) overhead in execution time.

of doitgen, as discussed before, nESLT backups too many elements and thus incurs higher overhead. For the same kernel, in case of oESLT, interestingly, we realize slight performance gains because of the backup code! This we believe is because of the possible cache benefits due to the backup code (it acts as a prefetch loop). To establish this hypothesis, we increased the input size for doitgen (from 256 to 384 = 16*24) and found that the benefits due to the backup loop decreases; the backup loop stops working as a prefetch loop and hence pollutes the cache.

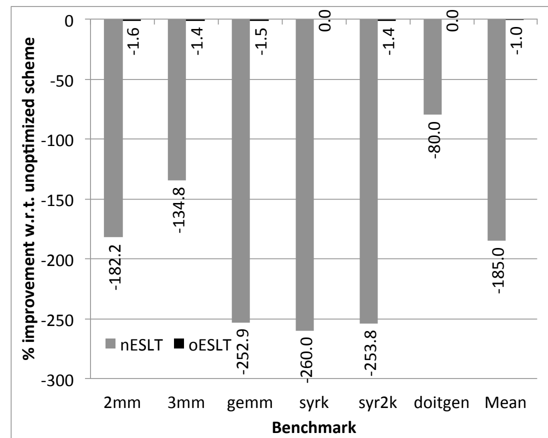
5.3 Impact of ESLT when exceptions are thrown

Figures 22 shows the impact of ESLT, when exceptions are thrown inside the tiled loop. Based on our experience with exceptions in OO programs that predominantly use exceptions to report corner cases (e.g., array being updated is null, the update to the array is not within the array bounds, and so on, which typically occur either at the beginning of the loop, or towards the end), we present a study of nESLT and oESLT, by forcing the exception to be thrown only in the first element of the first tile (Figure 21a), or last element of the last tile (Figure 21b) of the loop-nest.

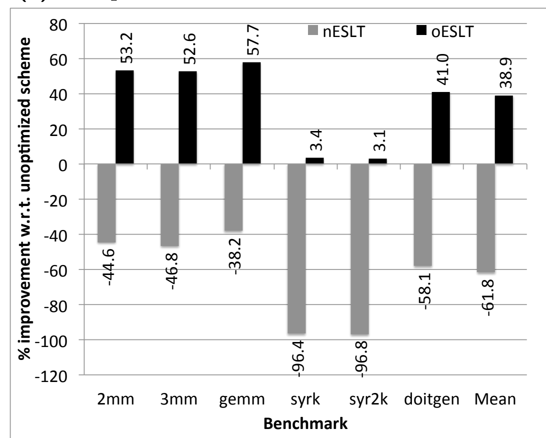
In Figure 21a, it can be seen that nESLT incurs significant performance degradation (ranging between 80.0% to 260.0%, geometric mean 185%) owing to the time spent in conservatively backing up all the array elements at the beginning. On the other hand, oESLT incurs a minor performance degradation (between 0.0% to 1.6%, geometric mean 1%) owing to the minimal backing up that is done as part of the optimized exception-safe loop tiling scheme.

In Figure 21b, it can be seen that nESLT again incurs significant performance degradation (between 38.2% to 96.8%, geometric mean 61.8%), owing to the time spent in conservative rollback and safe-execution of all the iterations. On the other hand, oESLT leads to significant improvements in performance (between 3.1% to 57.7%, geometric mean 38.9%). This is because the time savings resulting from tiling did offset the minimal time spent on rollback and safe execution.

Overall, it can be seen that oESLT leads to significant improvements when no exceptions are thrown (common case). In the not so common case, when an exception is indeed thrown, the gains depend on the exact iteration in which the exception is thrown (i.e., if the overheads are offset by the gains due to tiling).



(a) Exception thrown in the first tile.



(b) Exception thrown in the last tile.

Figure 22 Percentage (%) improvement in execution time.

Even though we use `throw` statements to represent the abnormal-exits, our proposed techniques can be used to handle any typical non-local control transfer statement (for example, goto, break, return, and so on). The techniques can also be applied on loops that contain multiple abnormal-exits of different types. Further, our techniques can handle conditional `throw` statements over both affine and non-affine conditions (a common scenario).

6 Conclusion and Future Work

In this paper, we present a generalized scheme to do exception-safe loop optimizations and present a scheme of optimized exception-safe loop tiling (oESLT), as a specialization thereof. oESLT leads to performance gains (because of loop tiling), with minimal overhead (due to backup and rollback).

Usually, the loops with exception `throw` statements (common in Java, C++) have multiple exit edges and to identify such loops in a general-purpose compiler (for example, GCC), we defined a new program region called ESCoP. We implemented our proposed techniques (built on top of ESCoPs) in GCC and show significant gains over the kernels from the PolyBench suite.

As a part of future work, we plan to understand the impact of the backup code on the tile size and the profitability. Designing an efficient exception-safe loop tiling scheme for parallel code is another interesting future work.

Acknowledgements. We would like to thank Shashidhar G for helping with the experimental setup, Rupesh Nasre and Raghesh Aloor for their insightful comments on a prior version of the manuscript and insightful discussions, in general. This research work is partially supported by the New Faculty Seed Grant, funded by IIT Madras (CSE/11-12/567/NFSC/NANV), the DRDO research grant (CSE/08-09/103/DRDO/HODX), and the DAE research grant (CSE/13-14/139/BRNS/NANV). We thank all these agencies for their generous funding and support.

References

1. M. Allen and S. Horwitz. Slicing java programs that throw and catch exceptions. In *PEPM*, pages 44–54, 2003.
2. R. Baghdadi, A. Cohen, S. Verdoolaege, and K. Trifunović. Improved loop tiling based on the removal of spurious false dependences. *TACO*, 9(4):52:1–52:26, 2013.
3. C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT*, pages 7–16, Juan-les-Pins, Sep 2004.
4. A. A. Belevantsev, S. S. Gaisaryan, and V. P. Ivannikov. Construction of speculative optimization algorithms. *Program. Comput. Softw.*, 34(3):138–153, 2008.
5. M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *CC*, LNCS, 2010.
6. C. Bienia, S. Kumar, J. Pal Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT*, pages 72–81, New York, NY, USA, 2008. ACM.
7. R. Bodík, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.
8. M. G. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *JAVA*, pages 129–141, 1999.
9. J. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. *SE Notes*, 24(5):21–31, 1999.
10. J. Collard. Automatic parallelization of while-loops using speculative execution. *International Journal of Parallel Programming*, 23(2):191–219, 1995.
11. P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.
12. K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), May 2007.
13. C. Fu and B. G. Ryder. Exception-chain analysis: Revealing exception handling architecture in Java server applications. In *ICSE*, pages 230–239. IEEE, 2007.
14. M. Gupta, J-D Choi, and M. Hind. Optimizing Java programs in the presence of exceptions. In *ECOOP*, pages 422–446. Springer-Verlag, 2000.
15. J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.
16. M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300. ACM, 1993.
17. IBM. XL C/C++ Compiler. `http://www-03.ibm.com/software/products/en/xlcpp-aix`.

18 K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, Implementation, and Evaluation of Optimizations in a Just-in-time Compiler. In *JAVA*, pages 119–128, 1999.
19 D. Jackson and E. J. Rollins. Chopping: A generalization of slicing. Technical Report CMU-CS-94-169, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.
20 R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *PLDI*, pages 171–185, 1994.
21 J. Lin, W. Hsu, P. Yew, R. D. Ju, and T. Ngai. Recovery code generation for general speculative optimizations. *TACO*, 3(1):67–89, 2006.
22 V. V. Mikheev, S. A. Fedoseev, V. V. Sukharev, and N. V. Lipsky. Effective enhancement of loop versioning in java. In *CC*, pages 293–306, 2002.
23 J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *TOPLAS*, 22(2):265–295, 2000.
24 S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
25 V. K. Nandivada and S. Jagannathan. Dynamic state restoration using versioning exceptions. *HOSC*, 19(1):101–124, 2006.
26 V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar. A Transformation Framework for Optimizing Task-Parallel Programs. *TOPLAS*, 35(1):3:1–3:48, April 2013.
27 L. N. Pouchet. PolyBench: The Polyhedral Benchmark suite.
28 L. Renganarayanan, D. Kim, M. M. Strout, and S. Rajopadhye. Parameterized loop tiling. *TOPLAS*, 34(1):3:1–3:41, May 2012.
29 H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao. Single-dimension software pipelining for multidimensional loops. *TACO*, 4(1), March 2007.
30 S. Sinha and M. J. Harrold. Analysis and testing of programs with exception-handling constructs. *IEEE TSE*, 26(9):849–871, September 2000.
31 Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *PLDI*, pages 215–228, New York, NY, USA, 1999. ACM.
32 R. M. Stallman and GCC DeveloperCommunity. *Using The GNU Compiler Collection: A GNU Manual For GCC Version 4.8.0*. CreateSpace, Paramount, CA, 2013.
33 S. Verdoolaege. *ISL*: An integer set library for the polyhedral model. In *ICMS*, pages 299–302, 2010.
34 T. Würthinger, C. Wimmer, and H. Mössenböck. Array bounds check elimination in the context of deoptimization. *Sci. Comput. Program.*, 74(5-6):279–295, Mar 2009.
35 J. Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, 2000.
36 H. Yun, J. Kim, and S. Moon. Optimal software pipelining of loops with control flows. In *ICS*, pages 117–128. ACM, 2002.

Transparent Object Proxies for JavaScript

Matthias Keil[1], Sankha Narayan Guria[2], Andreas Schlegel[1], Manuel Geffken[1], and Peter Thiemann[1]

1 **Institute for Computer Science**
University of Freiburg
Freiburg, Germany
{keilr,schlegea,geffken,thiemann}@informatik.uni-freiburg.de

2 **Indian Institute of Technology Jodhpur**
Jodhpur, India
sankha@iitj.ac.in

Abstract

Proxies are the swiss army knives of object adaptation. They introduce a level of indirection to intercept select operations on a target object and divert them as method calls to a handler. Proxies have many uses like implementing access control, enforcing contracts, virtualizing resources.

One important question in the design of a proxy API is whether a proxy object should inherit the identity of its target. Apparently proxies should have their own identity for security-related applications whereas other applications, in particular contract systems, require transparent proxies that compare equal to their target objects.

We examine the issue with transparency in various use cases for proxies, discuss different approaches to obtain transparency, and propose two designs that require modest modifications in the JavaScript engine and cannot be bypassed by the programmer.

We implement our designs in the SpiderMonkey JavaScript interpreter and bytecode compiler. Our evaluation shows that these modifications of have no statistically significant impact on the benchmark performance of the JavaScript engine. Furthermore, we demonstrate that contract systems based on wrappers require transparent proxies to avoid interference with program execution in realistic settings.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases JavaScript, Proxies, Equality, Contracts

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.149

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.1.1.2

1 Introduction

A proxy modifies the functionality of an underlying target object by introducing a level of indirection that intercepts all operations on the target. As all problems in computer science can be solved by another level of indirection[1], proxies may be called the Swiss army knives of object adaptation. Indeed, proxies are widely used to perform resource management, to access remote objects, to impose access control [20, 13], to implement contract checking [19, 10, 5], to restrict the functionality of an object [19], to enhance the interface of an object [22], to

[1] A famous quote by David Wheeler.

© Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann;
licensed under Creative Commons License CC-BY

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 149–173

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
1 var target = { /* some object */ };
2 var handler = { /* empty handler */ };
3 var proxy = new Proxy (target, handler);
4 proxy===target; // evaluates to false
```

Listing 1 Comparing a proxy with its target. The methods of *handler* determine the behavior of *proxy*. If *handler* is empty, then *proxy* behaves exactly like *target*.

implement dynamic effect systems, as well as for meta-level extension, behavioral reflection, security, and concurrency control [16, 2, 4].

A proxy implementation provides an intercession API that enables the programmer to trap all operations on the target object (with few exceptions). Further, a program should not be able to distinguish a proxy from a non-proxy object so that putting a proxy in place of an object does not affect the outcome of the program (save for the new behavior introduced by the proxy). For that reason, the JavaScript Proxy API [20], a part of the current ECMAScript 6 draft standard, does not include a function that checks whether an object is a proxy, it does not provide traps for all operations on objects, and it restricts some traps to avoid breaking certain object invariants [21].

However, the JavaScript Proxy API embodies a design decision that reveals the presence of proxies in some important use cases. This decision concerns object equality. The API description[2] says: *The double and triple equal (==, ===) operator is not trapped. p1 === p2 if and only if p1 and p2 refer to the same proxy.* The standard does not even mention proxies in the definition of object equality: *If x and y are the same Object value, return true.* [9, Section 7.2.13] In other words, proxies are *opaque*, which means that each proxy has its own identity, different from all other (proxy or non-proxy) objects. Given opaque proxies, an equality test can be used to distinguish a proxy from its target as demonstrated in Listing 1.

Even though *target* and *proxy* behave identically, they are not considered equal. Thus, in a program that uses object equality, the introduction of a proxy along one execution path may change the meaning of the program without even invoking an operation on the proxy (which may behave differently from the same operation on the target).

Equality for opaque proxies is straightforward to implement and works well under the assumption that proxies and their targets are never part of the same execution environment. For example, the revocable membrane pattern [20] enables to safely pass object references to untrusted code, control their operation on these objects, and revoke all access rights afterwards. This pattern is implemented using proxies and it partitions the execution environment into security realms so that the objects that live in the same realm are never in a proxy-target relationship. By this convention, the situation outlined in Listing 1 never arises inside a compartment.

But the assumption that proxies never share their execution environment with their targets is not always appropriate. One prominent use case is the implementation of a contract system. A contract system provides a domain specific language to state very precise type-like assertions for values in an untyped language. Two examples for such systems are the contract framework of Racket [11, Chapter 7] and Contracts.js for JavaScript [5]. Both systems implement contracts on objects with specific wrapper objects, Racket's chaperones or impersonators [19] and JavaScript proxies, respectively.

[2] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Proxy

```
1  // contract wrapper implementation
2  function checkPredicate (pred) {
3    return {
4      set: function(target, prop, val) {
5        if (!pred (val)) { throw new ContractException(); };
6        target[prop] = val;
7      }
8    }
9  };
10 function assertContract(target, pred) {
11   return new Proxy (target, checkPredicate(pred));
12 }
13 // application code
14 function addBonus (acc1, acc2, amount) {
15   acc1.balance += amount;
16   if (acc1 !== acc2) { // test objects for equality
17     acc2.balance += amount;
18   }
19 }
20 var account = { balance: 10 };
21 var restricted = assertContract (account, function(x) {
22   return (x >= 0);
23 });
24 addBonus (account, account, 40); // raises account by 40
25 addBonus (restricted, account, 40); // raises account by 80
```

Listing 2 Application with contract wrapper.

Listing 2 contains a JavaScript implementation of a simple contract wrapper. The implementation uses JavaScript proxies, which are introduced in more detail in Section 2. The wrapper applies to a JavaScript object and it enforces that all property values written to the object fulfill a predicate *pred*. Otherwise, the wrapper raises an exception.

The call *assertContract (target, pred)* returns a proxy for the target object with a handler created by the function call *checkPredicate (pred)*. Whenever a property *prop* is set on the proxy, the method *set* of its handler is invoked with the target object, the property *prop*, and the new value as arguments. The handler throws an exception if the predicate *pred* is not fulfilled, otherwise it performs the set operation on the target.

The application code contains an *addBonus* function that takes two accounts and a bonus amount to add. The intention is to give a bonus to each account once. Thus, if the two accounts are different, then the balance of the second account must be adjusted, too.

The last couple of lines create an account and a restricted handle to the same account, where the restricted handle does not permit the account to overdraw. In line 24, a bonus of *40* is added to *account* and *account*. This bonus addition executes correctly because the equality test in line 16 yields *false*. However, performing the bonus addition with the restricted version and the standard account leads to adding a bonus of *80* to *account* because the test in line 16 yields *true*.

This example shows that the introduction of a contract monitor like *assertContract* may change the semantics of a program even in cases where the contract is not exercised. But this

change in behavior violates a ground rule for monitoring: a monitor should never interfere with a program conforming to the monitored property. (In Section 3, we make a similar case for access restricting membranes.)

While this particular example is constructed we demonstrate in Section 7 that such situations do occur in practice. Racket programmers have also run into this issue[3], as chaperones and impersonators behave opaquely with respect to Racket's `eq?` operation.

As a remedy, we propose alternative designs for *transparent proxies* that are better suited for use cases such as certain contract wrappers and access restricting membranes. We evaluate these designs with respect to usability. We further implement them in the Firefox SpiderMonkey JavaScript engine and evaluate the impact of transparent proxies on benchmark performance.

Overview and contributions

Section 2 introduces the JavaScript Proxy API, explains the membrane pattern, and sketches the implementation of a contract system based on proxies. Section 3 discusses different use cases of proxies and assesses them with respect to the requirements on proxy transparency. Section 4 contains an in-depth discussion of the programmer's expectation from an equality operation and how it would be affected by the design of a proxy API. Section 5 presents alternative designs to obtain transparency. **We present two novel designs for transparent proxies that do not impede the implementation of advanced proxy management.** In Section 6, we describe how **we implement these two designs in the Firefox JavaScript engine (interpreter and baseline JIT compiler)**. In Section 7, **we demonstrate that our modification to the JavaScript engine does not affect benchmark performance**. Furthermore, **we present evidence that the danger of interference for a contract implementation based on opaque proxies is real**: it arises in instrumented benchmark programs, not just in artificially constructed examples. Section 8 discusses related work and Section 9 concludes.

The technical report [12] accompanying this paper contains an appendix with a detailed descriptions of the algorithms used in our implementation. The implementation of the JavaScript engine with transparent proxies is available in a Github repository[4].

2 Proxies, Membranes, and Contracts

This section introduces the JavaScript Proxy API and presents two typical applications of proxies: membranes that regulate access to an object network and contracts that check assertions on the values manipulated by a program.

2.1 Proxies

A proxy is an object intended to be used in place of a *target object*. As the target object may also be a proxy, we call the unique non-proxy object that is transitively reachable through a chain of proxies the *base target* for each proxy in this chain. The behavior of a proxy is controlled by a *handler object*, which may modify the original behavior of the target object in many respects. A typical use case is to have the handler mediate access to the target object, but in JavaScript the handler has a full range of intercession facilities that we only touch on.

[3] Personal communication with Robby Findler, February 2015.
[4] `https://github.com/sankha93/js-tproxy`

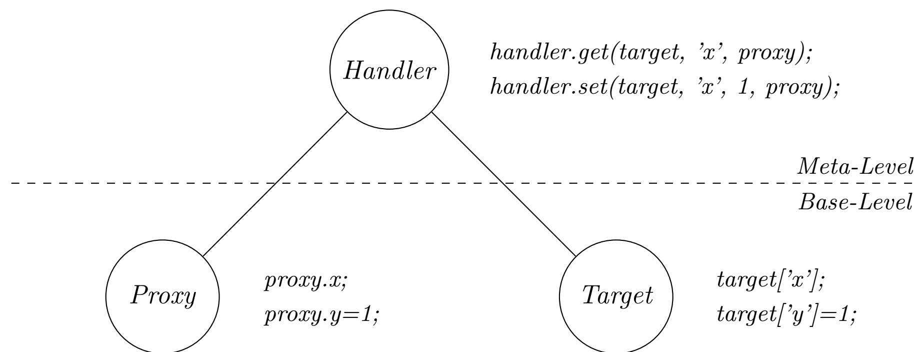


Figure 1 Example of a proxy operation.

The JavaScript Proxy API [20] contains a proxy constructor that takes the designated target object and a handler object:

```
26 var target = { /* some properties */};
27 var handler = { /* trap functions... */ };
28 var proxy = new Proxy (target, handler);
```

The handler object contains optional trap functions that are called when the corresponding operation is performed on the proxy. Operations like property read, property assignment, and function application are forwarded to the corresponding trap. The trap function may implement the operation arbitrarily, for example, by forwarding the operation to the target object. The latter is the default functionality if the trap is not specified.

Performing an operation like property get or property set on the proxy object results in a meta-level call to the corresponding trap on the handler object. For example, the property get operation *proxy.x* invokes *handler.get(target,'x',proxy)* and the property set operation *proxy.y=1* invokes the trap *handler.set(target,'y',1,proxy)*, if these traps are present.

Figure 1 illustrates this situation with a handler that forwards all operations to the target.

However, a handler may redefine or extend the semantics of an operation arbitrarily. For example, a handler may forward a property access to its target object only if the property is not locally present. The following example demonstrates a handler that implements a copy-on-write policy for its target by intercepting all write operations and serving reads on them locally. Thus, reading *target.a* at the end may return a different value than *42*.

```
29 function makeHandler() {
30   var local = {};
31   return {
32     get: function(target, name, receiver) {
33       return (name in local) ? local[name] : target[name];
34     },
35     set: function(target, name, value, receiver) {
36       return local[name]=value;
37     }
38   };
39 }
40 var child = new Proxy (target, makeHandler());
41 child.a = 42; // does not change target
```

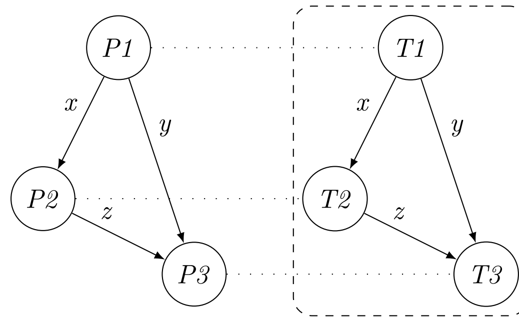


Figure 2 Property access through membrane.

Proxy and handler objects are based on the JavaScript Proxy API [20, 21], which is part of the JavaScript draft standard ES6. This API is implemented in Firefox since version 18.0 and in Chrome V8 since version 3.5.

JavaScript's proxies are *opaque*: each proxy object has its own identity different from all other (proxy) objects. The proxy identity is observable with the JavaScript equality operators == and ===: When applied to two objects, both operators compare object identities.[5]

The following example (which continues the preceding code fragment) illustrates this behavior. Comparing distinct proxies returns false even though the underlying target is the same. Similarly, the target object is different from any of its proxies.

```
42 var proxy2 = new Proxy (target, handler);
43 (proxy==proxy2); // false
44 (proxy==target); // false
```

We already mentioned in the introduction that equality cannot be trapped. While there are good reasons for this design decision [21], we mention in passing that it would be hard to design and implement an efficient equality trap because equality is a binary method.

2.2 Membranes

A *membrane* is a regulated communication channel between an object and the rest of the program. It ensures that all objects reachable from an object behind the membrane are also behind the membrane. Figure 2 shows a membrane (dashed line) around targets *T1*, *T2*, and *T3* implemented by the wrapper objects *P1*, *P2*, and *P3*. Each property access through the wrapper (e.g., *P1.x*) returns a wrapper for *T1.x*, which is created on demand. After installing the membrane, no *new* direct references to target objects behind the membrane become available. This mechanism may be used to revoke all references to an object network at once or to enforce write protection on the objects behind the membrane [20, 17].

An *identity preserving membrane* is a membrane that furthermore guarantees that no target object has more than one proxy. Thus, proxy identity outside the membrane reflects target object identity inside. That is, if *T1.x.z===T1.y*, then also *P1.x.z===P1.y*. Figure 2 depicts such an identity preserving membrane.

Both kinds of membrane may be implemented with the JavaScript Proxy API and a weak map that associates target objects with their proxies [20].

[5] If one argument has a primitive type, == attempts to convert the other argument to the same primitive type, whereas === returns false if the types are different. If both arguments are objects, then both operators do the same.

2.3 Contracts

Dynamically checked software contracts lie at the core of Meyer's *Design by Contract*™ methodology [15]. A contract specifies the interface of a software component by stating obligations and benefits for the component's implementors and users. Contracts state invariants for objects as well as preconditions and postconditions for functions and methods. Contracts are particularly important for dynamically typed languages as these languages do not provide static guarantees beyond memory safety. For such languages, contract systems are indispensable tools to create maintainable software.

A contract may specify a property of a value. In many cases, a simple assertion suffices, but many interesting properties of functions and objects cannot be checked immediately. For example, the contract $Num \to Num$ expresses that a function maps a number to a number. It can only be checked when the function is called: the caller must provide a number argument and then the result must be a number, too. Similarly, a check that a certain object property must always be assigned a number can only be checked when actually setting the property.

We call such contracts on functions and objects *delayed contracts*, because their assertion never signals a contract violation immediately. The standard implementation of a delayed contract is by wrapping the function/object in a proxy. The proxy's handler implements traps to mediate the use of the function/object and to assert its contract when the function is called or the object is read or written to.

The following example sketches the implementation of a contract assertion for $Num \to Num$. It installs a proxy where the handler supplies an *apply* trap that gets invoked when the proxy is called as a function. The arguments to *apply* are the target object, the ***this*** object, and an array containing the arguments.

```
45 var handler = {
46   apply: function(target, thisArg, argsArray) {
47     if (typeof argsArray[0] !== 'number') { throw new Exception (); }
48     var result = target.apply(thisArg, argsArray);
49     if (typeof result !== 'number') { throw new Exception (); }
50     return result;
51   }
52 };
53 var addOne = function (x) { return x+1; };
54 var addOneNN = new Proxy (addOne, handler);
```

Thus, the monitored function *addOneNN* is implemented by a proxy with a suitable handler. The contract systems Contract.js [5] and TreatJS [14] are both implemented in this way. As JavaScript does not support "proxification" (i.e., the transformation of an arbitrary object into a proxy), it is conceivable that *addOne*, the original object, and *addOneNN*, the proxy, are both accessible in the same execution environment.

3 Opacity vs. Transparency for Proxies

In this section, we question whether JavaScript proxies need be opaque by considering various use cases for proxies and evaluating whether they could be served equally well with transparent proxies, where equality is defined as equality of base targets.

3.1 Use Case: Object Extension

A common use case of proxies is to extend or redefine the semantics of particular operations on objects. For example, a handler may throw an exception instead of returning *undefined*, it may redirect different operations to different targets (for example to store changes locally or to implement placeholders), it may log or trace operations, or it may notify observers.

In this case, using the proxy may lead to a completely different outcome than using the target object. Thus, proxy and target object should not be confused.

3.2 Use Case: Access Control

Revocable references are the motivating use case for membranes [20, 17]. Instead of passing a target object to an untrusted piece of code, the idea is to pass its proxy wrapped in a protecting membrane. Once the host application deems that the untrusted code has finished its job, it revokes the reference which detaches the proxy from its target. The membrane extends this detachment recursively to all objects reachable from the original target.

Opaque proxies are suitable for implementing membranes as well as their identity preserving variant. However, transparent proxies would work just as well, because the host application only sees original objects whereas the untrusted code only sees proxies. Furthermore, the implementation of revocable references and membranes ensures that there is at most one proxy for each original object. If an execution environment is compartmentalized like this, then each compartment has a consistent view with unique object (or proxy) references, regardless whether proxies are opaque or transparent. In fact, with transparent proxies, a membrane is always identity preserving, the weak map only improves the space efficiency.

3.3 Use Case: Contracts

Proxies implement contracts in Racket [19] and in JavaScript [5, 13, 14]. During maintenance, a programmer may add contracts to a program as understanding improves. To systematically investigate a program in this way, the addition of a new contract must not change a program execution that respects the contract already. In this scenario, the program executes in a mix of original objects and proxy objects. Furthermore, there may be more than one proxy (implementing different contracts) for the same target. If introducing proxies affected the object identity, then some equality comparisons on objects (`eq?` in Racket and *==*, *!=*, *===*, or *!==* in JavaScript) would flip their outcome, thus changing the semantics.

Our experimental evaluation (Section 7.2) considers a typical program understanding and maintenance scenario where a programmer inserts assertions/contracts to document and validate his/her understanding of the program. We find that mixed (proxy vs. non-proxy) object comparisons occur in realistic programs.

Similar incidents where observed when maintaining Racket's preferences framework. Registering a callback with the framework wrapped the callback in a contract before storing it in an `eq?`-based weak hash table. Because there were no further references to the wrapper, the weak table released it on the next garbage collection. Thus, the callback disappeared mysteriously, leading to unwanted behavior[6]. This problem is evidence that such mixes occur in real situations and also in a system where contracts are aligned with module boundaries.

Thus, we see strong evidence that unintended mixing is hard to avoid even in a well-designed system. If a module has a higher-order interface, then a function passed as a

[6] Personal communication with Robby Findler, February 2015.

parameter may capture an un-proxied version of an object that is also passed as a regular parameter. For example, let T be some delayed contract and consider the module interface

```
55 // foo : (T → boolean, T) → boolean
```

so that *foo* carries a wrapper that asserts this contract. The function accepts two parameters, the first of which is a function. An external caller may use *foo* as follows:

```
56 var x = { /* some object */ };
57 var f = function (y) { return x===y; };
58 foo(f, x);
```

The call to *foo* wraps *f* and x in the respective contract wrappers for $T \rightarrow$ *boolean* and T. Unfortunately, wrapping the function *f* does not affect the free variable x in its environment.

Now consider the following contract-abiding implementation of *foo*:

```
59 function foo (f, y) { return f (y); }
```

Inside of *foo*, y is wrapped in the T contract and applying *f* (wrapped in $T \rightarrow$ *boolean*) may wrap it one more time in a T. Thus, in the body of *f* (line 57), x is unwrapped and y is the same object wrapped at least once in T. Thus, assuming opaque proxies, $x{=}{=}{=}y$ yields *false*, which is different from the result before installing the contract on *foo* (line 55).

Thus, the existing implementations of higher-order contracts for JavaScript are prone to interfere with the semantics. The situation is similar for Racket. Racket's chaperones and impersonators are opaque because they may be distinguished from their target and from one another using `eq?`. This choice is legitimized by pointing out that the preferred equality test in Racket is the `equal?` function that compares two values for *structural equality*, a non-trivial functionality that is not available out-of-the-box in JavaScript (cf. [1]). Clearly, chaperones and impersonators are transparent with respect to `equal?`. The paper on chaperones and impersonators [19] further acknowledges that "wrappings do affect the identity of objects, as compared with JavaScript's $===$ or Racket's `eq?` comparisons", but remarks that Racket programmers rely much less on object comparison than JavaScript programmers. Indeed, the paper's formalization includes `equal?`, but not `eq?`.

3.4 Assessment

Neither the opaque nor a transparent proxy implementation can be labeled as right or wrong without further qualification. Each is appropriate for particular applications and may lead to undesirable behavior in other applications.

Opaqueness is required for a proxy that changes the behavior of its target significantly. This case corresponds to the use case for impersonators [19].

Transparent proxies can safely be used to implement revocable membranes as well as for other applications that guarantee compartmentalized execution (where proxies and targets never meet). Their use simplifies the implementation of the identity preserving membrane because the weak map from targets to their proxies may be elided. However, the price for this elision is increased space usage for multiple wrappers for the same target. It must be weighed against the time taken to administer a weak target-to-proxy mapping.

Transparency is required for proxies that implement contract wrappers to avoid the interference with the normal program execution pointed out in the introduction and in Section 3.3. To avoid such interference with opaque proxies, a programmer would have to guarantee that contracts are only ever applied to unique object references or that references to the target object do not escape. Such a guarantee may be established by only applying

contracts to objects as they are created or by static analysis. However, the former is an unrealistic assumption and the latter severely limits the freedom that developers want to obtain by using contracts in a dynamically typed language: rather than submitting to a type system, they must submit to a uniqueness or an escape analysis.

A similar case can be made for further wrapper-based programming patterns. For example, a wrapper that records all operations performed on an object reference can be very helpful while debugging. Clearly, such a wrapper must be indistinguishable from its target object.

It is also clear that the behavior of equality is not something that should be left to the whim of the programmer. For example, equality on objects should be an equivalence relation, which means that the equality operations == and === must not be trapped [21].

Thus, the current state of affairs in JavaScript is fully justified, but it is not well suited to implement contract systems. To obtain some insight into a proxy design suitable for implementing contracts, we first explore the important issue of equality and then consider some designs and assess the suitability with respect to the use cases outlined in this section.

4 Invariants for Equality

What does a programmer expect from an equality function on objects? Let's consult the language references for Racket, Java, and JavaScript for cues.

Racket inherits its hierarchy of equality operators from Scheme: `equal?`, `eqv?`, and `eq?`. The Scheme report [18] specifies them as follows: "An equivalence predicate is the computational analogue of a mathematical equivalence relation; it is symmetric, reflexive, and transitive. Of the equivalence predicates described in this section, `eq?` is the finest or most discriminating, `equal?` is the coarsest, and `eqv?` is slightly less discriminating than `eq?`." The intuition is that `equal?` implements structural equality, `eq?` implements pointer equality, and `eqv?` is a compromise between the expensive structural equality[7] and pointer equality, which may distinguish different boxings of the same number.

There are further differences between `eq?` and `equal?`. The function `eq?` is guaranteed to run in $O(1)$, whereas there is no known implementation of `equal?` that runs in less than linear time. Furthermore, `equal?` is not stable in the sense that `(equal? x y)` may hold at some point during execution, but this equality may be destroyed by subsequent assignments in the program. In contracts, `(eq? x y)` does not change as the program proceeds.

Java has an `equals` method in `java.lang.Object`. The documentation of this class reads:[8] "The equals method implements an equivalence relation on non-null object references: . . . " It also asks that repeated invocations with the same arguments behave consistently in that they always return the same answer. And finally: "The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`)." About the `==` operator, the JLS says:[9] "The equality operators are commutative . . . " and then "the result of `==` is `true` if the operand values are both `null` or both refer to the same object or array; otherwise, the result is `false`." Regarding stability and execution time, the `==` operator is stable and runs in $O(1)$. The `equals` methods is recommended to be implemented in a stable way, but this restriction is not enforced. No bounds on the execution time are prescribed and none can be given.

[7] It is supposed to work on cyclic structures, too. [1]
[8] `http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html`
[9] `http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html` in 15.21

The ECMAScript specification [8, Section 11.9] does not mention algebraic properties of the equality operation, but rather contains pseudocode for the abstract equality comparison algorithm, which underlies the == operator. This algorithm implements an operation, which is symmetric, but neither reflexive nor transitive: It is not reflexive because *NaN==NaN* is *false* and it is not transitive because ***new*** *String("foo")=="foo"* and *"foo"==****new*** *String ("foo")*, but ***new*** *String("foo")==****new*** *String("foo")* is *false* due to unfortunate interaction with type conversions. Restricted to object arguments the algorithm says: "Return true if x and y refer to the same object. Otherwise, return false." The strict equality comparison algorithm (in Section 11.9.6), which specifies the === operator, is not reflexive because of *NaN*, but appears to be symmetric and transitive. Thus, it is at least a partial equivalence relation. Restricted to objects, both equalities are equivalences if that is implied by sameness.

JavaScript's == operator is not stable: Consider a comparison between an object and a string and then change the *toString* method of the object. However, === is stable because it does not involve type conversion. Restricted to object arguments, both equalities are stable.

The least common denominator for an object equality appears to be that **equality must be a stable equivalence relation**. Often, object equality is explained by alluding to the "sameness" of objects, which is left to further interpretation. In particular, the JLS explicitly mentions that `==` is overly discerning for `String` objects and none of the language specifications addresses proxy objects.

Some proponents of opaque proxies ask that equality should be an observational equivalence. That is, for any conditional of the form

```
60 if (a === b) Statement
```

(where *a* and *b* are variables) it should be possible to substitute *b* for *a* (but respecting lexical binding rules) in *Statement* without changing the semantics. However, in the presence of JavaScript's *with (head){ ... body ... }* statement inside *Statement* further qualifications are needed. As *with* extends the lexical environment by placing *head* on top of the scope chain while executing *body*, a property of *head* may shadow any variables in scope including *a*. Thus, the substitution must not extend to the body of a *with* statement.

If the above conditional appears in the *body* of a *with (head){ ... body ... }* statement, then essentially all bets are off. The object *head* may define *a* or *b* via a getter function or via a proxy handler, which may be impure and return different results on each access. Alternatively, *a* and *b* may be changed by simple assignment to *head*.

Furthermore, the *Statement* may contain a call to *eval*. As this call is executed in the lexical environment of its call site, its execution may perform an assignment to *a* or *b*. As the call to *eval* may be performed indirectly by storing *eval* in a different variable or object property, essentially every function or method call may assign to *a* or *b*.

While ES5 strict mode abolishes the *with* statement and severely restricts the use of *eval*, a call to *eval* may still assign to variables in scope at the point of the call. For example, the following function *f42* always returns *42* because the *eval* assigns to an existing local variable.

```
1 function f42(x) {
2   "use strict";
3   eval("x = 42’’);
4   return x;
5 }
```

Thus, arriving at a practically useful definition of observational equivalence in JavaScript is quite intricate; perhaps too intricate to be a useful reasoning tool for programmers.

Moreover, we are not aware of a language definition that requires its equality operator to be an observational equivalence.

What is the take-home message from this discussion with respect to the question whether a proxy implementation should be opaque or transparent with respect to equality? We believe that most programming patterns using equality only expect a stable equivalence relation, which is easy to implement for transparent proxies, as we show in Section 5.3. However, disregarding the problems with the *with* statement and *eval*, a weak version of observational equivalence is certainly desirable: For any conditional of the form

```
61 if (a === b) Statement
```

if we substitute *b* for *a* in *Statement*, then either the semantics of *Statement* does not change or *Statement* raises an exception.

However, a transparent proxy without further restrictions would not even fulfill weak observational equivalence. As a compromise, the behavioral change of a transparent proxy could be restricted to implement a *projection*, a point that we come back to in Section 5.4.

5 Design Alternatives for Proxy Equality

In this section, we explore some alternatives for designing proxies and equalities and discuss their suitability for the use cases outlined in Section 3.

5.1 Program Rewriting

One way to obtain transparent proxies with an implementation of opaque proxies is to replace all occurrences of *==*, *!=*, *===*, and *!==* with proxy-aware equality functions. These functions can be implemented in JavaScript using a weak map from proxies to their target. The proxy constructor would be extended to maintain this map. It would be possible to treat some proxies as transparent and the rest as opaque.

This approach preserves the existing behavior and retains the possibility to distinguish proxies from target objects in library code implementing proxy abstractions. A macro system like SweetJS [6] may be used to implement such a transformation elegantly, alas, SweetJS is currently implemented as an offline transformation and would need to be extended to deal with `eval` and dynamic loading.

5.2 Additional Equality Operators

Another approach would be to reinterpret the JavaScript equality operators == and === as proxy-transparent and introduce new variants, *:==:* and *:===:*, say, for their opaque cousins (i.e., the current implementations of == and ===). The former operators are used in application code whereas the implementation of proxy abstractions would make use of the opaque operators where needed.

No code transformation is required with this approach. However, it is not clear how to ensure that application code does not use the opaque operators. It is not even clear if it *should not* use them. While proxy abstractions can be implemented, the distinction between application and library seems too brittle.

Given both operations, application code can test if two objects are in a proxy relation with the same target:

```
62 ((x === y) != (x :===: y)); // true, iff x is in a proxy relation to y
```

```
1 function GetIdentityObject(obj) {
2   while(isProxy(obj) && isTransparentProxy(obj)) {
3     obj = getProxyTargetObject(obj);
4   }
5   return obj;
6 }
```

Listing 3 Pseudo code for GetIdentityObject.

Furthermore, the implementation would either have to use macros, which gives rise to the problems discussed in Section 5.1, or it would have to be implemented in the JavaScript engine, where it requires changes starting in the parser. This point makes it unlikely that such a proposal would be adopted by the community. Moreover, a proliferation of equality operations is confusing for developers as there are already three different kinds of equality in JavaScript: besides equality and strict equality, there is a third equality that fixes reflexivity when comparing *NaN* values.

5.3 Transparent Proxies in the VM

We already discussed that trapping the equality operation is not appropriate. As an alternative, we implement transparent proxies as an extension of a JavaScript VM (cf. Section 6) and provide different constructors for transparent and for opaque proxies. This extension provides a different kind of proxy object on which equality comparison behaves differently. Before testing reference identity as the last step in a comparison of two objects, the equality comparison calls a new internal function *GetIdentityObject* (see Listing 3) that computes the base target of an object. For a non-proxy object, the function returns its argument. For a proxy object, *GetIdentityObject* checks whether the proxy is transparent. If that check fails, then *GetIdentityObject* returns the reference to the current proxy object. Otherwise, it iteratively performs the same checks on the proxy's target. For consistency, the *GetIdentityObject* method also needs to be called in other computations that depend on object identity. One example is the WeakMap abstraction of the ES6 draft standard.

This design enables both scenarios described in Sections 3.2 and 3.3. It also guarantees that equality (on objects) is an equivalence relation.

Transparent proxies need special attention because there are abstractions that require to test whether a reference is a (transparent) proxy. For example, the implementation of access permissions contracts [13] extracts the current permission from a proxy to construct a new proxy with an updated permission. This introspection improves the efficiency of the implementation; its absence would lead to long, inefficient chains of proxy objects.

Thus, for implementing proxy abstractions it is useful to be able to break the transparency. We propose to use secret tokens for this purpose. A token (just an object in JavaScript) stands for a transferable right to perform a particular operation. We attach the token object to a proxy by making it an extra argument to the constructor of transparent proxies, *TransparentProxy*, say. Being a standard object, the token can be hidden in the scope of the function that wraps objects.

```
63 var wrap = (function() {
64   var token = {};
65   return function(target, handler) {
66     return new TransparentProxy(target, handler, token);
```

```
67   };
68   // further operations on wrappers
69 })();
```

Later on, the token can be used to make the transparent proxy visible for equality operations. To this end, we need an equality operation *Object.equals* that takes the token as a third (optional) parameter. The following snippet demonstrates this operation in action.

```
70 var token = {};
71 var target = { ... };
72 var proxyA = new TransparentProxy (target, handlerA, token);
73 var proxyB = new TransparentProxy (target, handlerB, token);
74 target == proxyA; // true
75 proxyA == proxyB; // true
76 Object.equals(target, proxyA, token); // false: token reveals proxy identity
77 Object.equals(proxyA, proxyB, token); // false
78 Object.equals(target, proxyA, { /* some other object */ }); // true
79 Object.equals(proxyA, proxyB, { /* some other object */ }); // true
```

Weak maps and other internal data structures that depend on object equality may be extended with tokens in the same way.

From different point of view, the tokens assign transparent proxies to distinct realms. Thus, instead of passing tokens one could use object capabilities to create proxies in a particular realm and to create an equality function that only reveals proxies for that realms[10]. A realm constructor may be implemented in JavaScript on top of our token-based implementation.

```
80 TransparentProxy.createProxyConstructor = function() {
81   var token = {};
82   var equals = function (x, y) {
83     return Object.equals(x, y, token);
84   }
85   var Constructor = function(target, handler) {
86     return new TransparentProxy(target, handler, token);
87   }
88   return {Constructor:Constructor, equals:equals};
89 }
```

The realm constructor returns a new transparency realm represented by an object that consists of a fresh constructor for transparent proxies (named *Constructor*) and an *equals* function revealing proxies of that realm.

```
90 var realm = TransparentProxy.createProxyConstructor();
91 var proxy1 == realm.Constructor(target, handler);
92 var proxy2 == realm.Constructor(target, handler);
```

The proxies *proxy1* and *proxy2* are transparent with respect to equality unless someone uses the *realm.equals* method.

```
93 proxy1 === proxy2; // true
94 Object.equals(proxy1, proxy2); // true
95 realm.equals(proxy1, proxy2); // false
```

[10] We would like to thank the anonymous ECOOP 2015 reviewer who suggested this elegant solution.

Here, === and *Object.equals* return *true* and do not reveal proxies. However, the *realm .equals* function is a capability that represents the right to reveal proxies of that realm. Realm-aware weak maps and other internal data structures can be created in same way.

5.4 Observer Proxies

To implement contracts, transparent proxies need not be fully general. It would be sufficient if transparent proxies would be restricted to *Observer*s that implement a projection: they would either return a value identical to the value that would be returned from the target object (this should also include the same side effects) or that restricts the behavior of the target object. An *Observer* can cause a program to fail more often, but otherwise it would behave in the same way as if no observer were present.

A similar feature is provided by Racket's chaperones [19]. A chaperone is a proxy that either returns an identical value, returns a chaperone of this value, or throws an exception. This restricted kind of proxy is shown to be sufficient to implement a contract system. (Recall that chaperones are not transparent.)

The following code snippet sketches the implementation of an *Observer* proxy in JavaScript that mimics Racket's chaperone proxy. The example considers the *get* trap only, but other traps can be implemented in the same way.

```
96  function Observer(target, handler) {
97    var sbx = new Sandbox(/* some parameters */);
98    var controller = {
99      /* further traps omitted */
100     get: function(target, name, receiver) {
101       var result = (trap=handler['get']) ? sbx.call(trap, target, name, receiver) :
            undefined;
102       var raw = target[name];
103       return (result===raw) ? result : raw;
104     }
105   };
106   return new TransparentProxy(target, controller);
107 }
108 var target = { /* some object */ };
109 var handler = {
110   get:function(target, name, receiver) {
111     return target[name];
112   }
113 };
114 var observed = Observer(target, handler);
```

The constructor starts with instantiating a sandbox (line 97). The sandbox is drawn from another contract system for JavaScript, TreatJS [14], that uses membranes and decompilation to implement access restrictions. Functions execute inside the sandbox without interfering with the normal execution.

The implementation distinguishes between a user specified handler (*handler*) whose traps are evaluated in the sandbox to guarantee noninterference and the proxy handler (*controller*) which is used to implement the behavior of the observer. The controller's *get* trap first checks the presence of a *get* trap in the user handler, before it evaluates this trap in the sandbox. Next, it performs a normal property access on the target value. This step is required to

produce the same side effects and to get a reference value to compare the results. Finally, the reference value is compared with the result from calling the trap. Line 103 makes only sense when using transparent proxies. The user specific trap can return an observer of the reference value.

Indeed, the implementation is only correct if one can ensure that *result* is either the *raw* value or a transparent proxy generated by an *Observer*. A user specific handler can simply elude the observers behavior by returning a transparent proxy of the same target but with a different handler object. Correctness can be guaranteed by either hiding transparent proxies from the user level or by using the sandbox to restrict resources access be the handler's trap.

5.5 Recommendation

There is likely no single semantics for object identity that fits the programmers expectation in all possible contexts. A proxy that changes the behavior of its target object significantly needs its own identity and thus needs to be implemented opaquely.

A contract proxy that only restricts the behavior of the original object, we propose to use a transparent observer proxy with the design explained in Section 5.3. For these proxies, === (and friends) will be forwarded to the target objects, recursively. An observer proxy (see Section 5.4) limits the possible change of behavior analogously to chaperones. Technically, observer proxy weakly simulate the original objects.

6 Implementation

We implemented two prototype extensions of the SpiderMonkey JavaScript engine, one according to the design of Section 5.3 and another which extends the proxy handler by an *isTransparent* trap that regulates the proxy's transparency. The first prototype implements a new global object *TransparentProxy* that implements the constructor for transparent proxy objects.

Proxies created with ***new*** *Proxy* in the second prototype are generally opaque, unless they implement an *isTransparent* trap and this trap returns false. Proxies created with ***new*** *TransparentProxy* are generally transparent unless they are compared with *Object.equal* using the correct token. These choices guarantee full backwards compatibility.

The implementation is rather tedious because many things have to be implemented three times as SpiderMonkey consists of an interpreter, the baseline JIT compiler (JaegerMonkey), and the IonMonkey compiler. Support for transparent proxies has to be added at each level because SpiderMonkey switches at run time from interpreter to baseline compiler and then to IonMonkey after a sufficient number of loop iterations. Specifically, the documentation says: "All JavaScript functions start out executing in the interpreter [... which] collects type information for use by the JITs. When a function gets somewhat hot, it gets compiled with JaegerMonkey. [...] When it gets really hot, it is recompiled with IonMonkey." If type information changes, execution falls back all the way to the interpreter.

6.1 JavaScript Interpreter

To cater for transparent proxies, the interpreter had to be changed in a few places.

1. The internal classes `Proxy` and `BaseProxyHandler` had to be extended to support the new *isTransparent* trap.
2. The comparison operators had to be modified to recognize transparent proxies and obtain their base target for comparison,

3. All internal data structures which are connected to the identity of objects (in particular, the map, weak map, and set abstractions of the upcoming JavaScript standard) had to be modified.

6.1.1 JavaScript's Equality Comparison

JavaScript provides two types of comparison operators. The *strict equality comparison* (e.g. ===) returns *false* if the operands have different types. The *equality comparison* (e.g. ==) applies type conversion if the operands have different types; then it essentially performs a strict comparison on the converted values. When comparing two objects x and y, both comparisons behave identically [8, Section 11.9.3]:

> 1.f. "Return *true* if x and y refer to the same object."

Thus, this test for sameness of two objects is only one place where the algorithms for equality comparison and strict equality comparison have to be changed. Our implementation replaces the test (case 1.f. in equality comparison [11.9.3], case 7. in strict equality comparison [11.9.6]) as follows.

1. Let *lhs* be the result of calling *GetIdentityObject* on x.
2. Let *rhs* be the result of calling *GetIdentityObject* on y.
3. Return *true* if *lhs* and *rhs* refer to the same object. Otherwise, return *false*.

6.1.2 Getting the Identity Object

When comparing two objects or when adding an object to a map, transparent proxies do not use their own identity. To get the right identity for the object the operation first checks the transparency of the proxy object and transitively obtains its target object until either an opaque proxy or a native object is reached (cf. Listing 3 for the pseudocode). All object comparisons refer to this internal method.

The actual implementation is slightly more involved, in particular the implementation that supports the *isTransparent* trap (its existence needs to be checked, it needs to be called, and its results needs to be interpreted). Technical details may be checked in the source code which is available in a github repository.

6.1.3 Maps, Sets, and other Data Structures

After modifying the comparison operators the internal data structures *Map*, *Set*, and *WeakMap*, which depend on object equality, have to be adjusted to handle transparent proxies. If *target == proxy* evaluates to *true* then *map.has(target)==map.has(proxy)* should also evaluate to *true*.

When adding a new key-value pair to any Map, WeakMap, or Set, the operation first determines if the key is an object of type *Proxy*. If it is a *Proxy*, then the *GetIdentityObject* internal method is used to determine the identity object; hashing takes place with respect to this identity object, but the original key is stored in the collection along with its value. A subsequent lookup of the identity object or any proxy with the same identity object returns the same stored value. The implementation of the ***for...in*** loop or calling *.keys()* or *.entries()* on a map returns the originally added object as key value.

The example below demonstrates the behavior just described on an empty map object *map*. We first create one transparent and one opaque proxy for the same target. The operation *map.set(target, A)* creates a new map entry, whereas the second one *map.set(proxy1, B);* updates this entry. The third operation *map.set(proxy2, C);* creates a new entry, again.

```
115 var target = {};
116 var proxy1 = new TransparentProxy(target, {});
117 var proxy2 = new Proxy(target, {});
118 map.set(target, A); // map = [target ↦ A]
119 map.set(proxy1, B); // map = [target ↦ B]
120 map.set(proxy2, C); // map = [target ↦ B, proxy2 ↦ C]
```

6.2 Object.equals

Transparent proxies created with *TransparentProxy* are generally indistinguishable from their base target object and from another transparent proxy object of the same target. However, *Object.equals* can be employed to make them distinguishable for algorithms that implement advanced proxy manipulation. To this end, the constructor stores its token argument in a slot of each proxy.

When *Object.equals* is called with arguments *obj1*, *obj2* and an optional argument *secret* the following steps are taken:

- If *secret* is not present, then return the value of *obj1 === obj2*.
- If one of *obj1* or *obj2* is a transparent proxy where the token slot matches the *secret*, then return *true* if *obj1* is the same object as *obj2* (and *false*, otherwise).
- Otherwise return the value of the transparent comparison *obj1 === obj2*.

6.3 JavaScript Baseline Compiler

The SpiderMonkey Baseline Compiler is the first tier of the JIT compiler. It produces native code for JavaScript through stub method calls and optimized inline caches (ICs) for some operations. To adapt for changes to the equality (both strict and non-strict) comparison operation, the fallback stub code was modified to do a call to the VM and test for equality between the two objects in exactly the same manner as in the interpreter.

For the *isTransparent* trap implementation we stop emitting the optimized ICs for object-object comparison, and instead use the fallback IC to do a VM call. This will invoke the *isTransparent* trap for the proxy and then compare with the identity object if it evaluates to *true* or it performs the standard object-object comparison.

For the *TransparentProxy* implementation, we stop emitting the optimized IC for any object-object comparison that involves comparison of *TransparentProxy* object. We use the fallback stub to do a VM call and do a comparison with the identity object, if it involves a *TransparentProxy*. Any other kind of comparison operations are left unaffected and still take place through the optimized stubs.

6.4 IonMonkey Compiler

The IonMonkey optimizing compiler is the final tier of the SpiderMonkey JIT compiler. This has been left unmodified, but we give an outline for a future implementation so that generated native code by IonMonkey can support transparent proxy comparisons.

For the *isTransparent* trap implementation, it will need to load the object into the register and test if the object's class is a *Proxy* or not. If it is not a proxy, then normal fast path for object-object comparison code can be generated. Otherwise execution should stop, do a VM call to the *isTransparent* trap in the handler object of the proxy, and store the return value in a register. If the value is *false* then proceed with emitting the usual object comparison operation code, otherwise load the identity object of the proxy (i.e., the result of calling

GetIdentityObject on the proxy) and replace that value in the equality operation's operand register. Then we can proceed with emitting of code for a standard object comparison.

For the *TransparentProxy* implementation, the code generation for the equality operation would be simpler, as there is no handler trap to be called. We'd have to load the object into the register and test if the object's class is a *TransparentProxy* or not. If it is not a *TransparentProxy*, then normal fast path for object-object comparison code can be generated. Otherwise load the result of calling *GetIdentityObject* on the proxy and replace that value in the equality operation's operand register. Then we can proceed with emitting of code for a standard object comparison.

6.5 Getting the Source Code

The implementation of both modified engines is available on the Web[11]. The branch *isTransparent-trap*[12] contains the isTransparent handler trap version and branch *global-tproxy-object*[13] contains the implementation of the TransparentProxy object. A *README* file in each of the respective branches contains the build instructions.

7 Evaluation

This section reports our experiences with applying our modified engines to JavaScript benchmark programs to answer the following research questions:

RQ1 Does the introduction of transparent proxies affect the performance of non-proxy code?

RQ2 Does a contract implementation based on opaque proxies affect the meaning of realistic programs?

7.1 Performance Test

To answer **RQ1**, we evaluate the impact of our implementations of transparent proxies on programs that do not make use of proxies at all. These programs may be affected by our modifications to the equality comparison algorithms and to the set and map abstraction.

To this end we used benchmark programs from the Google Octane 2.0 Benchmark Suite[14]. This suite comprises 17 programs that range from performance tests to real-world web applications (Figure 3), that is, from an OS kernel simulation to a portable PDF viewer. Each program focuses on a special purpose, for example, function and method calls, arithmetic and bit operations, array manipulation, JavaScript parsing and compilation.

Octane reports its results in terms of a score. The Octane FAQ[15] explains the score as follows: "*In a nutshell: bigger is better. Octane measures the time a test takes to complete and then assigns a score that is inversely proportional to the run time.*" The constants in this computation are chosen so that the current overall score (i.e., the geometric mean of the individual scores) matches the overall score from earlier releases of Octane and new benchmarks are integrated by choosing the constants so that the geometric mean remains the same. The rationale is to maintain comparability.

[11] `https://github.com/sankha93/js-tproxy/`
[12] `https://github.com/sankha93/js-tproxy/tree/isTransparent-trap`
[13] `https://github.com/sankha93/js-tproxy/tree/global-tproxy-object`
[14] `https://developers.google.com/octane/`
[15] `https://developers.google.com/octane/faq`

Benchmark	Origin		Trap		Transparent	
	No-Ion	No-JIT	No-Ion	No-JIT	No-Ion	No-JIT
Richards	505	64.8	502	63.3	509	64.3
DeltaBlue	453	82.5	466	80.2	466	79.6
Crypto	817	111	825	113	793	109
RayTrace	462	182	455	173	462	174
EarleyBoyer	909	275	938	271	913	270
RegExp	853	371	842	362	871	365
Splay	802	409	792	398	857	409
SplayLatency	1172	1336	1222	1307	1231	1338
NavierStokes	841	155	836	156	834	148
pdf.js	2759	704	2764	697	2793	691
Mandreel	691	82.5	711	82.4	688	78.5
MandreelLatency	3803	526	3829	514	3829	503
Gameboy Emulator	4275	556	4250	535	4382	540
Code loading	9063	9439	9124	9318	9114	9502
Box2DWeb	1726	289	1750	278	1736	282
zlib	28981	29052	29097	29074	28909	29108
TypeScript	3708	1241	3715	1210	3666	1203
Total Score	1594	456	1604	447	1610	445

Figure 3 Scores for the Google Octane 2.0 Benchmark Suite (bigger is better). Column **Origin** gives the baseline scores for the unmodified engine. Column **Trap** shows the score values of the engine that implements the transparency trap and column **Transparent** contains the scores for running the engine containing the transparent proxies. The sub-column **No-Ion** (*no IonMonkey*) lists the scores with the baseline compiler enabled, but with IonMonkey disabled. Sub-column **No-JIT** (*no just in-time compilation*) shows the scores of the interpreter without any kind of just in-time compilation.

7.1.1 The Testing Procedure

All benchmarks were run on a machine with two AMD Opteron processors with 2.20 GHz and 64 GB memory. All measurements reported in this paper were obtained with *SpiderMonkey JavaScript-C24.2.0*.

To evaluate our implementation we run the benchmark program in different settings to separate the impact caused by the interpreter and the baseline compiler. Recall that no modifications were done to IonMonkey. Enabling IonMonkey in this state would lead to meaningless results from executing a mixture of proxy-aware code and proxy-oblivious code. Therefore, **the IonMonkey compiler remains disabled**.

7.1.2 Results

Figure 3 contains the score values of all benchmark programs in different configurations explained by the figure's caption. The examples show the run-time impact of our modified engines vs. an unmodified engine. All scores were taken from a deterministic run, which requires a predefined number of iterations, and by using a warm-up run.

Comparing the total scores of the interpreter (column **No-JIT**), the **Trap** version is 1.97% slower than the unmodified engine and the **Transparent** version is 2.41% slower than the unmodified engine. However, when comparing the total scores of the baseline compiler (column **No-Ion**) we see that the **Trap** version is 0.62% faster than the unmodified engine and that the **Transparent** version is 1.00% faster than the unmodified engine.

At this point we have to mention that both differences are smaller than the standard deviation of the mean total scores produced by an unmodified engine. When measuring five runs with the same configuration we found a standard deviation of 2.68 score points for the interpreter and a standard deviation of 23.44 points for the baseline compiler.

The numbers clearly show that both implementations do not have a statistically relevant impact on the execution time of non-proxy code. This result is not surprising because the overwhelming majority of equality comparisons have at least one non-object operand. As our modification only applies when both operands are objects it is only exercised rarely (cf. Section 6) and hence its performance impact is not measurable.

7.2 An Analysis of Object Comparisons

In this section, we answer **RQ2**, by considering how many object-object comparisons occur during a normal program execution and how many of these may fail when objects were wrapped by contracts implemented using opaque proxies. To this end, we count object comparisons involving JavaScript Proxies and give a classification that covers different types of object comparisons whose result might be influenced by the transparency of the involved proxy objects.

7.2.1 The Testing Procedure

For this experiment, we instrumented the JavaScript engine with a monitor to count and classify object-object comparisons. Our subject programs are again taken from the Google Octane 2.0 Benchmark Suite. Our wrapper model is a simple contract system which applies a dynamic type check (cf. Section 2.3) to the arguments of selected functions.

To prepare for the experiment, a source-to-source translation generates for each function that occurs in a program a new variant of the program, where exactly this function is replaced by a function that wraps its arguments with a transparent proxy. The proxy's handler implements a membrane. It forwards the operation to the target and wraps its return value in another transparent proxy. We rely on a weak map to avoid creating chains of nested proxies.

We applied this translation to the benchmark programs and executed each variant in our new engine, counting and classifying each object comparison. As we used transparent proxies, normal execution of the programs is not influenced. Because each function is wrapped individually, we can accurately detect the effect of each single placement of a contract.

7.2.2 Results

First we introduce the types of object comparisons that must be distinguished. We only consider comparisons between two objects where at least one of them is a proxy object, because these are the only comparisons that may be affected if the proxy is opaque.

Type-I All comparisons between a proxy object and another object, which is either a native object or a proxy object from another membrane. Comparisons of this type always return *false* when using opaque proxies. With transparent proxies the result is either *true* or *false*, depending on the proxy's target object.

Type-Ia The subset of **Type-I**, where the underlying target objects differ. Opaque and transparent proxies yield the same outcome, *false*, but for different reasons.

Table 1 Number of comparisons involving object proxies. Column **Total** contains the total number of comparisons. Column **Type-I** lists the comparisons of **Type-I**, divided in the two categories **Type-Ia** and **Type-Ib**. Column **Type-II** shows the number of **Type-II** comparisons, divided in the categories **Type-IIa** and **Type-IIb**.

Benchmark		Type-I		Type-II	
	Total	Type-Ia	Type-Ib	Type-IIa	Type-IIb
DeltaBlue	144126	29228	1411	33789	79698
RayTrace	1075606	0	0	722703	352903
EarleyBoyer	87211	8651	6303	53389	18868
TypeScript	801436	599894	151297	20500	29745

Type-Ib The subset of **Type-I**, where the underlying target objects are the same. A comparison of this type yields *false* when using opaque proxies, whereas transparent proxies yield *true*.

Type-II All comparisons between two proxy objects from the same membrane. If using an identity preserving membrane, a target object is only wrapped once. In this case the result will be *true* if and only if they refer to the same target object, independent of the transparency of the proxies involved. Without an identity preserving membrane, the use of opaque proxies yields *false*, whereas the result with transparent proxies depends on the proxy's target object.

Type-IIa The subset of **Type-II**, where the target objects differ. Opaque and transparent proxies yield the same outcome, *false*, but for different reasons.

Type-IIb The subset of **Type-II**, where both proxies refer to the same target object. A comparison of this type yields *false* when using opaque proxies without an identity preserving membrane, whereas transparent proxies or an identity preserving membrane yield *true*.

In this setting we count the comparison between two proxy objects from different membranes in category **Type-I**, because different contracts implement different membranes and the mechanism that preserves the identity does not work when using different membranes.

Clearly, the **Type-Ib** comparisons are the bad guys as they may flip. They are closely followed by **Type-IIb** comparisons, although they are avoidable if identity preserving membranes are used throughout.

Table 1 summarizes the number of comparisons between native objects and proxy objects and among proxy objects. Comparisons between two primitive values (e.g., a boolean, a number, a string, null, or undefined), comparisons between a primitive value and an object (proxy or native object), and comparisons between two native objects are omitted from the results because the result of the operation is not influenced by the transparency of proxy objects.

Benchmark programs not listed in this table do not contain comparisons with a proxy object: any function in the unlisted programs may be wrapped using any kind of membrane without affecting its meaning. However, they still contain comparisons between native objects and primitive values (usually the test *ptr === null*).

The numbers in the table cover all comparison operators, namely *equal* (==), *not equal* (!=), *strict equal* (===), and *strict not equal* (!==). The meaning of "the result is *true*" is generalized to the sense that *equal* and *strict equal* will return *true*, *not equal* and *strict not equal* will return *false*.

What we see in Table 1 is that there are three benchmarks with a non-negligible number of bad **Type-Ib** comparisons, although the majority of object comparisons is not affected. The numbers also indicate that there are many more **Type-IIb** comparisons, so that any use of non-identity preserving membranes should be strongly discouraged.

7.3 Summary and Threats to Validity

The evaluation shows that the implementation of a dynamic contract system based on opaque proxies, whose monitoring replaces function arguments by proxy objects, definitely influences the program execution (which answers **RQ2**), which in turn leads to program errors.

The reason for the comparatively small number of flipped comparisons is due to the careful handling of object comparisons in JavaScript. Results from previous unpublished experiments show that approximately 6% of all comparisons involve two objects. The vast majority of comparisons either check an object against null or undefined, or compare primitive values.

8 Related Work

The JavaScript proxy API [20, 21] enables a developer to enhance the functionality of objects easily. The implementation of proxies opens up the means to fully interpose all operations on objects including functions calls on function objects.

JavaScript proxies have been used for Disney's JavaScript contract system, contracts.js [5], to enforce access permission contracts [13], as well as for other dynamic effects systems, meta-level extension, behavioral reflection, security, or concurrency control [16, 2, 4].

Proxy-based implementations avoid the shortcomings of static implementations and offline code transformations. In JavaScript, static approaches are often lacking because of the dynamicity of the language. Proxies guarantee full interposition and handle the full JavaScript language, including the *with*-statement, *eval*, and arbitrary dynamic code loading techniques.

The ideal contract system should not interfere with the normal execution of code as long as the application code does not violate any contract. The application should run as if no contracts were present [7].

Object equality becomes an issue for non-interference when contracts are implemented by some kind of wrapper. The problem arises if an equality test between wrapper and target or between different wrappers for the same target returns false instead of true. This issue is known from other work involving wrappers for implementing object extensions and multimethods [22, 3]

The PLT group examines various designs for low-level mechanisms for implementing contracts and related abstractions [19]. They propose two kinds of proxies, chaperones and impersonators, that differ, for example, in the degree of freedom for modifying the underlying object. They experience similar problems with noninterference as we report in Section 3.3 when using the *eq?* operator which is similar to JavaScript's strict equality operator $===$ and roughly implements pointer equality on objects. Racket's chaperones and impersonators are not transparent with respect to this operator. However, the preferred equality operation in Racket, *equal?*, implements structural equality which is indifferent to proxy transparency. In contrast, JavaScript provides no built-in operation to test for structural equality, so that developers need to build on pointer equality or roll their own structural equality.

9 Conclusion

Neither the transparent nor the opaque implementation of proxies is appropriate for all use cases. We discuss several amendments and propose two flexible solutions that enable applications requiring transparency as well as opacity. Both solutions are implemented as extensions of the SpiderMonkey JavaScript VM. This approach ensures full and transparent operation with all JavaScript programs. Hence, we can evaluate the solution on real-world JavaScript programs.

A significant number of object comparisons would fail when mixing opaque proxies and their target objects. This situation can arise when gradually adding contracts to a program during debugging. Identity preserving membranes decrease this number, but they are not able to guarantee full noninterference.

We also measured the run-time impact of an implementation supporting transparent proxies on the execution time of a realistic program mix. The results show that the modification to equality required to support transparent proxies has no statistically significant impact on the execution time.

References

1. Michael D. Adams and R. Kent Dybvig. Efficient nondestructive equality checking for trees and graphs. In Peter Thiemann, editor, *Proceedings International Conference on Functional Programming 2008*, pages 179–188, Victoria, BC, Canada, September 2008. ACM Press, New York.
2. Thomas H. Austin, Tim Disney, and Cormac Flanagan. Virtual values for language extension. In Cristina Videira Lopes and Kathleen Fisher, editors, *OOPSLA*, pages 921–938, Portland, OR, USA, 2011. ACM.
3. Gerald Baumgartner, Martin Jansche, and Konstantin Läufer. Half & half: Multiple dispatch and retroactive abstraction for Java. Technical Report OSU-CISRC-5/01-TR08, Revised 3/02, Ohio State University, March 2002.
4. Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA*, pages 331–344. ACM, 2004.
5. Tim Disney. contracts.js. `https://github.com/disnet/contracts.js`, April 2013.
6. Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. Sweeten your JavaScript: Hygienic macros for ES5. In Andrew P. Black and Laurence Tratt, editors, *DLS*, pages 35–44, Portland, OR, USA, October 2014. ACM.
7. Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal higher-order contracts. In Olivier Danvy, editor, *Proceedings International Conference on Functional Programming 2011*, pages 176–188, Tokyo, Japan, September 2011. ACM Press, New York.
8. ECMAScript Language Specification, December 2009. ECMA International, ECMA-262, 5th edition.
9. ECMAScript Language Specification. `http://wiki.ecmascript.org/lib/exe/fetch.php?id=harmony:specification_drafts&cache=cache&media=harmony:ecma-262_6th_edition_final_draft_-04-14-15.pdf`, April 2015. ECMA International, ECMA-262, 6th edition (draft).
10. Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In Simon Peyton-Jones, editor, *Proceedings International Conference on Functional Programming 2002*, pages 48–59, Pittsburgh, PA, USA, October 2002. ACM Press, New York.
11. Matthew Flatt, Robert Bruce Findler, and PLT. *The Racket Guide*, v.6.0 edition, March 2014. `http://docs.racket-lang.org/guide/index.html`.

12 Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent object proxies for JavaScript. Technical report, Institute for Computer Science, University of Freiburg, 2015.

13 Matthias Keil and Peter Thiemann. Efficient dynamic access analysis using JavaScript proxies. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS'13, pages 49–60, New York, NY, USA, 2013. ACM.

14 Matthias Keil and Peter Thiemann. TreatJS: Higher-order contracts for JavaScript. `http://proglang.informatik.uni-freiburg.de/treatjs/`, 2014.

15 Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

16 Mark S. Miller, Tom Van Cutsem, and Bill Tulloh. Distributed electronic rights in JavaScript. In Matthias Felleisen and Philippa Gardner, editors, *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 1–20, Rome, Italy, March 2013. Springer.

17 Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006. AAI3245526.

18 Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten, editors. *Revised[6] Report on the Algorithmic Language Scheme*. Cambridge University Press, 2010.

19 T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: Run-time support for reasonable interposition. In Gary T. Leavens and Matthew B. Dwyer, editors, *OOPSLA*, pages 943–962. ACM, 2012.

20 Tom Van Cutsem and Mark S. Miller. Proxies: Design principles for robust object-oriented intercession APIs. In William D. Clinger, editor, *DLS*, pages 59–72. ACM, 2010.

21 Tom Van Cutsem and Mark S. Miller. Trustworthy proxies – virtualizing objects with invariants. In Giuseppe Castagna, editor, *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 154–178, Montpellier, France, July 2013. Springer.

22 Alessandro Warth, Milan Stanojevic, and Todd Millstein. Statically scoped object adaptation with expanders. In *Proceedings of the 21th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 37–56, Portland, OR, USA, 2006. ACM Press, New York.

A Theory of Tagged Objects

Joseph Lee[1], Jonathan Aldrich[1], Troy Shaw[2], and Alex Potanin[2]

1 **Carnegie Mellon University**
Pittsburgh, PA, USA
`josephle@andrew.cmu.edu, aldrich@cs.cmu.edu`

2 **Victoria University of Wellington**
New Zealand
`troyshw@gmail.com, alex@ecs.vuw.ac.nz`

Abstract

Foundational models of object-oriented constructs typically model objects as records with a structural type. However, many object-oriented languages are class-based; statically-typed formal models of these languages tend to sacrifice the foundational nature of the record-based models, and in addition cannot express dynamic class loading or creation. In this paper, we explore how to model statically-typed object-oriented languages that support dynamic class creation using foundational constructs of type theory. We start with an extensible tag construct motivated by type theory, and adapt it to support static reasoning about class hierarchy and the tags supported by each object. The result is a model that better explains the relationship between object-oriented and functional programming paradigms, suggests a useful enhancement to functional programming languages, and paves the way for more expressive statically typed object-oriented languages. In that vein, we describe the design and implementation of the Wyvern language, which leverages our theory.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases objects, classes, tags, nominal and structural types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.174

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.1.1.3`

1 Introduction

Many models of statically typed object-oriented (OO) programming encode objects as records, usually wrapped inside a recursive or existential type [5]. These models elegantly capture many essential aspects of objects, including object methods, fields, dispatch, and subtyping. They are also foundational in that they describe core object-oriented constructs in terms of the fundamental building blocks of type theory, such as functions, records, and recursive types.

These models have been used to investigate a number of interesting features of OO languages. For example, dynamically-typed languages often support very flexible ways of constructing new objects and classes at run time, using ideas like mixins [10]. Typed, record-based models of object-oriented programming can support these flexible composition mechanisms with small, natural extensions [27].

Unfortunately, there are also important aspects of common object-oriented programming languages that are not adequately modeled by record-based object encodings. Many object-oriented languages are class-based: each object is an instance of a class, and classes are

© Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin;
licensed under Creative Commons License CC-BY

related by a subclass (or inheritance) relationship. In statically-typed class-based languages, subtyping is generally not structural, but instead follows the subclassing relation. Most class-based languages also provide a way for programmers to test whether an object is an instance of a class, for example via a cast, which results in an error if the test fails, or an `instanceof` operation, which returns a boolean.

Typical models of these languages, such as Featherweight Java [18], assume that each object is a member of a class, and that a fixed class table exists mapping classes to their definitions (and thus defining a subclassing relation). Casts or `instanceof` tests can then be encoded using class table lookups. Such dynamic instance checks are useful in cases where a programmer wishes to express the behavior of a sum type in a statically-typed OO language, as in abstract syntax tree evaluation. Scala makes the relationship between sum-types and instance checking explicit through its pattern-matching expression, which can pattern match an object against multiple possible subclasses [8].

The class-based models of object-oriented programming with which we are familiar have two limitations. The first limitation is the fixed nature of the class table: the models assume that the entire class table can be computed at compile time, meaning that run-time loading, generation, or composition of classes is not modeled. These models, therefore, do not capture the dynamic compositions that are possible in class-based dynamic object-oriented languages such as Smalltalk. As a result, they are an unsuitable foundation for important areas of research, such as exploring future statically-typed object-oriented languages that support more dynamic forms of composition.

The second limitation is that these class-based models are not foundational in nature, in that they do not explain classes and instance checking in terms of the fundamental constructs of type theory. The more ad-hoc nature of these models inhibits a full understanding of the nature of classes and how classes relate to ideas in other paradigms, such as functional programming. For example, as described above, Scala's support for case classes that mirror datatypes in functional programming suggests an analogy between OO class types and type theory sums, and between instance checking in object-oriented languages and pattern matching or tag checking in functional languages. In particular, the Standard ML language provides extensible sums through its `exn` type. This poses a natural question: could an `exn`-like construct, suitably extended, be used to model `instanceof` checks in object-oriented languages?

The primary contribution of this paper is exploring a more flexible and foundational model of class-based object-oriented programming languages. At a more detailed level, the contributions include:

- Section 2 defines a source-level statically-typed object-oriented language that supports dynamic creation of class hierarchies, including mixin composition, along with a match construct that generalizes instanceof checks and casts. We also show examples demonstrating the expressive power of the language, including a technique for enforcing encapsulation, the memento design pattern, and type-safe dynamic mixin composition.
- Section 3 defines a foundational calculus, consisting of a standard core type theory with the addition of constructs to support hierarchical tag creation, tag matching, and the extraction of tagged content. We define the static and dynamic semantics of the foundational language and prove soundness.
- In Section 4, the semantics of the source-level language is defined by a translation into the target language, thus explaining how dynamic, class-based OO languages can be modeled in terms of type theory. The translation motivates the particular tag manipulation constructs of the foundational calculus, explaining how constructs such as the `exn` type in

$$e ::= \texttt{class}\ x\ \{\overline{f:\tau}, \overline{m:\tau=e}\}\ \texttt{in}\ e \mid \texttt{class}\ x\ \texttt{extends}\ x\ \{\overline{f:\tau}, \overline{m:\tau=e}\}\ \texttt{in}\ e$$
$$\mid \texttt{new}(x;\bar{e}) \mid e.f \mid e.m \mid \texttt{match}(e_1;x_1;x_2.e_2;e_3) \mid \lambda x{:}\tau.e \mid e\ e \mid \texttt{let}\ x=e\ \texttt{in}\ e$$
$$\mid \texttt{letrec}\ x=e\ \texttt{in}\ e \mid x$$

$$\tau ::= \texttt{class}\ x\ \{\overline{f:\tau}, \overline{m:\tau}\} \mid \texttt{class}\ x\ \texttt{extends}\ x\ \{\overline{f:\tau}, \overline{m:\tau}\} \mid \tau \to \tau \mid x\ \texttt{obj}$$

Figure 1 Source Language Syntax.

Standard ML need to be adapted to serve as a foundation for object-oriented programming.
- In Section 5, we present the publicly available implementation of our language.

After presenting these contributions, our paper discusses extensions in Section 6 and additional related work in Section 7. Section 8 concludes with a discussion of the potential applications of the model.

2 First-Class Classes

We motivate our work with a source language that treats classes as first-class expressions, putting them in the same syntactic category as objects, functions, and other primitive forms. This language is designed to be as simple as possible in order to focus on the creation and usage of classes and objects. Figure 1 shows our source language syntax, which includes class definition, instantiation with `new`, method and field definition and selection, a `match` construct that generalizes `instanceof` and casts, the lambda calculus, `let`, and `letrec`. We omit mutable fields, inheriting method bodies, and other constructs in order to focus on the basic object-oriented mechanisms of classes, dispatch, and instance checks. The omitted constructs can be added in standard ways [25].

The most central construct of our language is the class definition expression, which defines the set of fields and methods as usual, and binds the newly defined class to a name x that can be used in the expression e. This binding is recursive so that methods can refer to the type of the containing class. We bind a name to the class immediately for this purpose and so that the type system can use the class's name to reason about its identity. However, class creation is dynamic: class definition expressions may occur inside a function or method, and a new class with a distinct run-time tag is created each time the function is called. The created class is also first-class: it has type $\texttt{class}\ x\ \{\overline{f:\tau}, \overline{m:\tau}\}$ and can be passed to a function or method, or bound to another variable y. A singleton type mechanism [17] could be added to support strong reasoning about equality of classes, but for simplicity we do not explore this here.

We illustrate the language with a simple counter class that increments a value when the `inc` method is called:

```
1 class Counter {
2   val:int,
3   inc:unit->Counter obj=λ_:unit.new(Counter;this.val+1)
4 } in let oneC = new (Counter; 1)
5   in let twoC = oneC.inc() ...
```

In the example, `val` is a field that will be initialized when a `Counter` is created. `inc` is a method; the difference between methods and fields is that a method is given a definition in the class body (typically using a function, although our semantics does not enforce this), and the special variable `this` is in scope within the body of the method.

As mentioned earlier, classes and objects in the source language are immutable for the sake of simplicity. Therefore, instead of actually updating the `val` field, the `inc` method

simply returns a new object with its `val` field equal to the receiver object's `val` field plus one. The return type of the `inc` method is `Counter obj`, which is the type of objects that are instances of the `Counter` class.

A subclass can be created using a variant of the class expression that specifies the superclass x. We require the use of a variable rather than an expression in the `extends` clause to allow static reasoning about the identity of the class being extended. A subclass can define additional members, or define members that are subtypes of the corresponding members in the superclass, thus following typical depth and width subtyping rules. The type of a subclass includes the name of the class it extends.[1]

To continue the simple counter example, consider a possible subclass that can reset the counter to zero:

```
1 class RCounter extends Counter{
2   ...
3   reset:unit->RCounter obj=λ_:unit.new(RCounter;0)
4 } in e
```

The $\texttt{new}(x; \overline{e})$ expression takes a variable x representing a class and a list of expressions. Executing the `new` expression instantiates an object of class x with the input expressions as initial values for the fields of the object. Alternative constructors with pre-defined field values can be created by wrapping `new` in a function or method.

An important piece of the source language is the `match` expression. This checks whether or not the object passed in as the first argument is an instance of the class named in the second argument. If the instance check passes, then the variable x_2 is bound to the object originally passed in, but x_2 is typed according to the class that was checked against, and expression e_2 is executed. Expression e_3 is executed in the default case where the match fails.

The match expression could be expressed equivalently in terms of `instanceof`, downcasting, and an `if` statement:

$$\begin{array}{l}\texttt{match}(e_1; x; y.e_2; e_3) = \\ \quad \texttt{if}(e_1 \texttt{ instanceof } x) \quad \texttt{then let } y = (x)e_1 \texttt{ in } e_2 \quad \texttt{else } e_3\end{array}$$

Similarly both `instanceof` and downcasting can be emulated by `match`:

$$\begin{aligned} e \texttt{ instanceof } x &= \texttt{match}(e; x; _.\texttt{true}; \texttt{false}) \\ \texttt{cast}(e; x) &= \texttt{match}(e; x; y.y; \texttt{err}) \end{aligned}$$

where `err` indicates an execution error (e.g. throwing an exception) upon a failed cast. The `err` construct demonstrates that `match` is safer than downcasts because a default case must be given.

Because class creation expressions can be nested in the body of a function or method, they can be executed more than once – and each time the expression is executed, a different class (i.e. a class with a different static type and run-time tag) is generated. This means that `match` cannot be implemented using a lookup in a static class table. Instead, as the next section will show, we need a dynamic environment tracking all the classes that have been created, along with their relationships to one another, in order to define the semantics of `match`.

[1] Although our system supports subtyping, we omit inheritance (e.g. of code in method bodies) in order to focus our attention on a simple type-theoretic formalism that captures the essence of tags.

2.1 Applications of First-Class Classes

There are a number of useful applications of first-class classes (and thus of dynamically generated hierarchical tags), a few of which we sketch here.

Encapsulation. One of the simplest applications is ensuring that data private to an object stays private. For example, imagine a map made up of a linked list of entries. We would like to ensure that no entry is shared by multiple maps. We can do this by generating a fresh `Entry` class for each map object. This idiom is similar to uses of generative functors in languages such as Standard ML [16].[2] In our language, it looks like this:

```
1  let MyMap =
2    class Entry {
3      key:int,
4      value:int,
5      next:Entry obj
6    } in
7      class Map extends IMap {
8        head:Entry obj,
9        put: int->int->Map obj = e_put,
10       get: int->int = e_get
11     } in Map
12 in ...
```

Each time the `let` statement is executed, we generate a new `Map` class, and each `Map` class has a corresponding `Entry` class used to hold its private data. The type system will not allow one map to contain entries defined by another map. Because all maps implement the same `IMap` interface, however, clients can store and manipulate maps uniformly.

Memento. A similar approach can be used to implement the Memento design pattern [12], in which an originator object externalizes its state as a separate memento object. The originator object's state can later be restored using the memento. We may want to ensure that mementos created by different objects are not mixed up. To do this, we generate a fresh `MyMemento` class for each originator object, ensuring that it is a subclass of the well-known `Memento` class. To externalize its state, each originator then creates memento objects using its own private `MyMemento` class. When the originator's state is restored, we check that the memento used for restoration is of the private class, ensuring that the memento was created for this object and not some other object. This example differs from the previous one in that mementos are deliberately exposed to clients, and in that checking for the right class is done dynamically using match rather than statically using the object type:

```
1  class Memento {intState:int} in
2  // the code below may be in a loop or function
3  let Originator =
4  class MyMemento extends Memento
5    {intState:int}
6   in
7    class Originator {
8     myState:int
9     saveToMemento: T -> Memento obj =
10      λx:int.new(MyMemnto; this.myState),
11    restoreFromMemento:Memento obj->Originator =
12      λm:Memento obj.
13        match(m;MyMemnto;
14          x.new(Originator;x.intState);
15          raise RuntimeException)
16   in Originator
17 in ...
```

[2] An alternative would be to use ownership types [7].

Mixins. Another application area for dynamically created tags is mixin compositions. According to Bracha and Cook, "A *mixin* is an abstract subclass; i.e. a subclass definition that may be applied to different superclasses to create a related family of modified classes. For example, a mixin might be defined that adds a border to a window class; this mixin could be applied to any kind of window to create a bordered-window class" [4].

Achieving the full benefits of mixins – and indeed, properly implementing the missing method bodies below – requires inheritance as well as tagged objects. Prior work on mixins shows how to obtain expressive code reuse via inheritance [2]. Our work is not intended to replace this work, but rather complement prior results by adding tagging support in a setting with first-class classes. In this section, we omit inheritance –which can be added back through standard encodings – in order to focus on the benefits of these added features.

The library code below illustrates a version of the bordered window example. Class Window defines a draw method (and perhaps other methods, not shown). We define a mixin as a function[3] that accepts a subclass of Window and extends that class with border functionality. A similar mixin can make an arbitrary window scrollable.

```
1  class Window {
2      draw:unit->unit = /* draw a blank window */
3  }
4
5  let makeBordered =
6      λwc: class WinClass extends Window { ... } .
7          class BorderedWinClass extends wc {
8              draw:unit->unit = /* draw a bordered window */
9          } in BorderedWinClass
10
11 let makeScrollable =
12     λwc: class WinClass extends Window { ... } .
13         class ScrollableWinClass extends wc {
14             draw:unit->unit =
15                 /* draw scrollbars and scroll contents */
16         } in ScrollableWinClass
```

Now, in the application code below, we can create a custom application window as a subclass of Window. We can then make it bordered, conditional on whether the user's preferences specify a border. This can be a run-time check, illustrating the dynamic nature of class composition in this example. We can also mix in the scrollable property, which will be used when creating large windows. Later, we can define operations such as screen capture that should behave differently for different kinds of windows, using a match statement.[4]

```
1  class AppWindow extends Window { ... }
2
3  let WinCls =
4      if (/* check if user wants a border */)
5          makeBordered(AppWindow)
6      else
7          AppWindow
8
9  let BigWinCls = makeScrollable(winCls) in
10 let smallWin = new WinCls(...) in
11 let bigWin = new BigWinCls(...) in
12 let screenCap = λw:Window .
13     match(w,BigWinCls,
```

[3] Similar to ML functor [16].

[4] We could also have made `screenCap()` a method of `Window`, but only if we anticipated it in the library. If we only think of it when we write the application, it is typically not feasible to add new methods to the library, and so using **`match`** statements in place of dispatch is critical.

$$n ::= x \mid c \mid \texttt{fst}(n) \mid \texttt{unfold}(n)$$

$$\begin{aligned} e ::= {} & \texttt{newtag}[\tau] \mid \texttt{subtag}[\tau](n) \mid \texttt{new}(n; e) \mid \texttt{match}(e; n; x.e; e) \mid \texttt{extract}(e) \mid \lambda x{:}\tau.e \mid e\ e \\ & \mid \{\overline{f = e}\} \mid e.f \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{letrec}\ x = e\ \texttt{in}\ e \mid \texttt{fold}[t.\tau](e) \mid \texttt{unfold}(e) \\ & \mid \langle e, e \rangle \mid \texttt{fst}(e) \mid \texttt{snd}(e) \mid \langle\rangle \mid n \end{aligned}$$

$$\tau ::= Tag(\tau) \mid \texttt{tagged}\ n \mid \Pi_{x:\tau}.\tau \mid \{\overline{f : \tau}\} \mid \mu t.\tau \mid \Sigma_{x:\tau}\tau \mid \top \mid t$$

$$Tag(\tau) ::= \tau\ \texttt{tag} \mid \tau\ \texttt{tag extends}\ n$$

$$\Gamma ::= \epsilon \mid \Gamma,\ x : \tau$$

$$\Sigma ::= \epsilon \mid \Sigma,\ c \sim \tau$$

$$\Delta ::= \epsilon \mid \Delta,\ t <: t$$

Figure 2 Core Language Syntax.

```
14          bw./*adjust capture for scrolling windows*/),
15          /*default screen capture*/)
16 in screenCap(bigWin); /* the match above succeeds */
17 screenCap(smallWin) /* the match above will default*/
```

Discussion: modules and dynamic class loading. The mixin example above is reminiscent of functors in Standard ML [16]. Functors can also be used to implement a type that builds on another type, which is bound later when the functor is applied. This suggests that our theory can be used to model the combination of classes and advanced module systems – and in fact, our implementation in Wyvern, discussed in Section 5, combines these features. We believe that our theory might also be useful for exploring the semantics of dynamic class loading, for similar reasons, but we have not yet investigated this connection.

3 A Hierarchical Tagging Language

We would like to find a way to explain the semantics of the language defined in the previous section in terms of type theory. Figure 2 defines the syntax of a core language for doing so. We start with constructs required for traditional object encodings based on recursive records: the lambda calculus, record creation and projection, ordinary and recursive let bindings, and iso-recursive types with fold and unfold constructs. Our function types are dependent so that we can define functions that act like functors, accepting a tag as an argument and producing a subtag of the argument as a result. We include units and a top type.

We also include dependent sum types, because our translation for the source-level language will translate a class into a pair containing the class's tag and a function to construct objects of that class. Since objects of the class have the class's tag as part of type, the constructor function must also; thus the pair has the type of a dependent sum.

These constructs are more than sufficient to encode record-based objects. However, we need more to allow us to model the classes and `match`-based instance testing constructs of the OO source language, while also supporting a type safety property.

As described in the introduction, there is a natural parallel between class types, class generation and match in our surface language, and extensible sums, tag generation and tag testing in functional programming languages such as Standard ML. We therefore add mechanisms for generating tags, tagging values, testing against tags, and extracting values from

tagged objects. Our presentation is inspired both by Glew's work on modeling hierarchical tags [13], and by the coverage of symbols and dynamic classification in Harper's book [14].

The most basic way to create a tag is using the expression $\texttt{newtag}[\tau]$ that, when executed, will generate a new tag (let's call it by the name n). The new tag can be used to create a *tagged value* using the expression $\texttt{new}(n; e)$, which takes the value to which e reduces and tags it with the tag n. In this expression, e must have the type τ that was associated with tag n when n was created. The type of the tagged expression is $\texttt{tagged}\ n$; thus the static type system tracks not only the fact that the expression is tagged, but what tag it is tagged with. In this respect, we model OO type systems that track the class of objects, but differ from prior work on modeling tags because they did not include the tag as part of the type [13, 14, 16].

While analogies can be drawn between tag creation and class creation, the type of τ is not restricted to be a record as in the source language. Rather, any arbitrary type τ is allowed to be tagged.

Following Glew [13], we provide a construct for extending an already created tag with a subtag. The construct $\texttt{subtag}[\tau](n)$ creates a new tag as a subtag of n. Note that tags to be extended in this rule are represented by names n; most prominently, names include variables x in the source code. Using names supports static reasoning about tag identity, just as in the source language we required the subclass to be specified as a variable. In addition to variables, names can be the first element of a dependent pair that is also represented by a name (we use this in our encoding – in principle we could include the second element as well), or an unfolding of a name, or they can be tag values c generated at run time. Here c is like a location in formalisms of references, and similar to locations, it cannot appear in the source code. We use Σ to associate tag values with types ($c \sim \tau$); this is analogous to a store type. Names *cannot* be arbitrary expressions; this restriction means we do not have to evaluate expressions in the type system, avoiding the intractability that this entails in the presence of recursion.

Tag testing is done through the $\texttt{match}(e_1; n; y.e_2; e_3)$ expression, which has the same structure as the `iftagof` construct from prior work in type theory [16, 13]. This expression takes in a tagged expression e_1, reduces it to a tagged value, and compares it to the tag n. Upon a successful match, the tagged value is bound to y, and the expression e_2 can use y as if it were of type $\texttt{tagged}\ n$. Like its source level equivalent, `match` requires there to be a failure branch, e_3, that is evaluated upon an unsuccessful tag test.

To extract an expression from a tag, $\texttt{extract}(e)$ is used. If e is a value tagged with n, then $\texttt{extract}(e)$ unwraps the tagged value and exposes the internal contents e.

Hierarchical tagging can be used to emulate a makeshift sum type. For example, the sum type $\texttt{int} + \top$, which would correspond to nullable `int`s, can be emulated in a local scope through tags:

$$
\begin{array}{l}
\texttt{let intOption} : \top\ \texttt{tag} = \texttt{newtag}[\top]\ \texttt{in} \\
\texttt{let none} : \top\ \texttt{tag extends intOption} = \texttt{subtag}[\top](\texttt{intOption})\ \texttt{in} \\
\texttt{let some} : \texttt{int tag extends intOption} = \texttt{subtag}[\texttt{int}](\texttt{intOption})\ \texttt{in} \\
\texttt{let}\ z : \texttt{tagged}\ \mathit{none} = \texttt{new}(\texttt{none}; \langle\rangle)\ \texttt{in} \\
(\lambda x : \texttt{tagged intOption}. \\
\quad \texttt{match}(x; \texttt{some}; y.\texttt{extract}(y) + 1; -1))(z)
\end{array}
$$

This expression would test `none` against `some`, and failing to match, will evaluate to -1. This example demonstrates how these constructs allow one to create extensible sum types in a local scope.

$$\frac{}{\Delta|\Gamma \vdash_\Sigma \tau <: \tau}(\text{ST-Reflexive}) \qquad \frac{\Delta|\Gamma \vdash_\Sigma \tau_1 <: \tau_2 \qquad \Delta|\Gamma \vdash_\Sigma \tau_2 <: \tau_3}{\Delta|\Gamma \vdash_\Sigma \tau_1 <: \tau_3}(\text{ST-Trans})$$

$$\frac{t <: t' \in \Delta}{\Delta|\Gamma \vdash_\Sigma t <: t'}(\text{ST-Amber-1}) \qquad \frac{\Delta, t{<:}t' \mid \Gamma \vdash_\Sigma \tau <: \tau'}{\Delta|\Gamma \vdash_\Sigma \mu t.\tau <: \mu t'.\tau'}(\text{ST-Amber-2})$$

$$\frac{}{\Delta|\Gamma \vdash_\Sigma \{f_i : \tau_i{}^{i\in 1..n+k}\} <: \{f_i : \tau_i{}^{i\in 1..n}\}}(\text{ST-Record-1})$$

$$\frac{\text{for each } i \quad \Delta|\Gamma \vdash_\Sigma \tau_i <: \tau_i'}{\Delta|\Gamma \vdash_\Sigma \{f_i : \tau_i{}^{i\in 1..n}\} <: \{f_i : \tau_i'{}^{i\in 1..n}\}}(\text{ST-Record-2})$$

$$\frac{\{l_j : \rho_j{}^{j\in 1..n}\} \text{ is a permutation of } \{f_i : \tau_i{}^{i\in 1..n}\}}{\Delta|\Gamma \vdash_\Sigma \{l_j : \rho_j{}^{j\in 1..n}\} <: \{f_i : \tau_i{}^{i\in 1..n}\}}(\text{ST-Record-3})$$

$$\frac{\Delta|\Gamma \vdash_\Sigma \tau_3 <: \tau_1 \qquad \Delta|\Gamma, x : \tau_3 \vdash_\Sigma \tau_2 <: \tau_4}{\Delta|\Gamma \vdash_\Sigma \Pi_{x:\tau_1}.\tau_2 <: \Pi_{x:\tau_3}.\tau_4}(\text{ST-App})$$

$$\frac{\Gamma \vdash_\Sigma n : \tau \texttt{ tag extends } n'}{\Delta|\Gamma \vdash_\Sigma \texttt{tagged } n <: \texttt{tagged } n'}(\text{ST-Tag-1})$$

$$\frac{\Delta|\Gamma \vdash_\Sigma \texttt{tagged } n <: \texttt{tagged } n'}{\Delta|\Gamma \vdash_\Sigma \tau \texttt{ tag extends } n <: \tau \texttt{ tag extends } n'}(\text{ST-Tag-2})$$

$$\frac{}{\Delta|\Gamma \vdash_\Sigma \tau \texttt{ tag extends } n <: \tau \texttt{ tag}}(\text{ST-Tag-3})$$

Figure 3 Core Language Subtyping Rules.

3.1 Subtyping

In our source language, subclasses define subtypes, and so our core language needs to define subtyping as well. The subtyping judgment is of the form $\Delta|\Gamma \vdash_\Sigma \tau <: \tau$. The extra context Δ for the subtyping judgment stores subtyping relations between type variables and is used in the Amber Rule (ST-Amber-1 and ST-Amber-2) for subtyping of recursive types [6].

Most of the subtyping rules in Figure 3 are standard: subtyping is reflexive and transitive. Recursive types follow the Amber Rule, while records support width and depth subtyping and allow permutation. Finally, we use the standard contravariant rule for dependent function types.

Subtyping between tagged types is nominal, and follows the declared subtagging relationship as expected: that is, `tagged` n is a subtype of `tagged` n' if the type of n reveals that it is a direct subtag of n'. We get transitivity of subtyping on tagged types from the general subtype transitivity rule.

The most interesting rules define the subtyping relation between the types of first-class tags themselves (as opposed to tagged types). If we have a function that accepts a first-class tag parameter of type τ `tag extends` n', what tags do we want it to accept? Intuitively, it should be acceptable to pass an actual argument of type τ `tag extends` n, where n is a subtag of n'; this loses a bit of information, but does not create any unsafe situations. Furthermore, if the function accepts a parameter of type τ `tag` (i.e. a tag with no known supertag) it should be fine to pass it an actual argument that *does* have a supertag, because none of the operations in our core language depend on a tag having no supertag.

On the other hand, it would be unsound to allow the type being wrapped by the tag to vary. This is because tags are like references: values flow into a tagged value when it is created, and they flow out when the `extract` operation is used, so the types of first-class tags cannot be either covariant or contravariant in the type being wrapped.

We can now see how to express the essence of the mixin example shown earlier can be expressed using the core language:

$\texttt{let}\ \mathit{base} : \texttt{int tag} = \texttt{newtag}[\texttt{int}]\ \texttt{in}$

$\texttt{let}\ \mathit{mixin} : \Pi_{c:\texttt{int tag}}.\texttt{int tag extends}\ c = \lambda c\colon \texttt{int tag}\ .\ \texttt{subtag}[\texttt{int}](c)\ \texttt{in}$

$\texttt{let}\ \mathit{sub} : \Pi_{c:\texttt{int tag}}.\texttt{int tag extends}\ c = \lambda c\colon \texttt{int tag}\ .\ \texttt{subtag}[\texttt{int}](c)\ \texttt{in}$

$\texttt{let}\ c : \texttt{int tag extends}\ \mathit{base} = \texttt{if}\ (\mathit{cond})\ \mathit{sub}(\mathit{mixin}(\mathit{base}))\ \texttt{else}\ \mathit{sub}(\mathit{base})\ \texttt{in}$

$\texttt{let}\ x : \texttt{tagged}\ c = \texttt{new}(c; 5)\ \texttt{in}$

$\ldots$

Here we have a tag *base* and two functions that create subtags of it, one of which is intended to be used as a mixin. Based on some dynamic condition *cond*, the program will create the tag c to either be a direct subtag of base or a transitive one. Because of the subtyping rules between tags, we can pass either *base* or $\mathit{mixin}(\mathit{base})$ to *sub* safely. Of course, the mixin above adds neither functionality nor state to the object, but the point of the example is getting the tags and the typing right; adding a standard encoding of inheritance will allow the mixin functionality to work.

3.2 Typing

The typing judgment for the core language is of the form:

$$\Gamma \vdash_{\Sigma} e : \tau$$

This judgment uses two different contexts. Γ is the standard type context, while Σ is introduced for technical reasons: it is like a store type, but rather than track the types of references, it tracks the types of generated tag values at run time, facilitating the proof of type safety (see the supplementary material [20]) in the same way that store types do in formalizations of imperative languages.

The typing rules for our core language are mostly standard; Figure 4 shows the rules for the constructs we introduced, as well as a couple of others that are more involved. The remaining rules are the standard ones for the lambda calculus, and so are omitted here. Our function application rule is the standard one, except that since we have a dependent function type, we must substitute the actual argument for the formal parameter in the result type. Since names, but not arbitrary expressions, can appear in types, applying the substitution to the result type is only defined when the argument e_2 is a name, or else when the variable x being substituted for is not free in the type τ' (note that if substitution is not defined then the App rule may not be applied).

Our rules include a subsumption rule that can be applied at any point, and a rule that looks up the type of a tag value c in the tag store type Σ (see the dynamic semantics, below, for how tag values are generated). Defining a new tag for τ values yields the obvious type τ `tag`. The rule for creating subtags is similar, but the premises must check that the name n given as a supertag is actually a tag, and that the type τ' being tagged is a subtype of the type τ that is tagged by the supertag. The `new` expression is well-typed as tagged by n if n is a tag and the expression used is a subtype of the type associated with the n tag.

$$\frac{\Gamma \vdash_\Sigma e_1 : \Pi_{x:\tau}.\tau' \qquad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma e_1 e_2 : [e_2/x]\tau'}(\textsc{App}) \qquad \frac{\Gamma \vdash_\Sigma e : \tau \qquad \epsilon|\Gamma \vdash_\Sigma \tau <: \tau'}{\Gamma \vdash_\Sigma e : \tau'}(\textsc{Sub})$$

$$\frac{\Sigma(c) = \tau}{\Gamma \vdash_\Sigma c : \tau}(\textsc{CVar}) \qquad \frac{}{\Gamma \vdash_\Sigma \texttt{newtag}[\tau] : \tau\ \texttt{tag}}(\textsc{Cls-I})$$

$$\frac{\Gamma \vdash_\Sigma n : \mathit{Tag}(\tau) \qquad \epsilon|\Gamma \vdash_\Sigma \tau' <: \tau}{\Gamma \vdash_\Sigma \texttt{subtag}[\tau'](n) : \tau'\ \texttt{tag extends}\ n}(\textsc{CCls-I})$$

$$\frac{\Gamma \vdash_\Sigma n : \mathit{Tag}(\tau) \qquad \Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \texttt{new}(n; e) : \texttt{tagged}\ n}(\textsc{Tag-I})$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma e_1 : \texttt{tagged}\ n' \qquad \Gamma \vdash_\Sigma \texttt{tagged}\ n' \updownarrow \texttt{tagged}\ n \\ \Gamma, x : \texttt{tagged}\ n \vdash_\Sigma e_2 : \tau \qquad \Gamma \vdash_\Sigma e_3 : \tau\end{array}}{\Gamma \vdash_\Sigma \texttt{match}(e_1; n; x.e_2; e_3) : \tau}(\textsc{Match})$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{tagged}\ n \qquad \Gamma \vdash_\Sigma n : \mathit{Tag}(\tau)}{\Gamma \vdash_\Sigma \texttt{extract}(e) : \tau}(\textsc{Extract})$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau_1 \qquad \Gamma \vdash_\Sigma e_2 : [e_1/x]\tau_2}{\Gamma \vdash_\Sigma \langle e_1, e_2 \rangle : \Sigma_{x:\tau_1}\tau_2}(\Sigma\text{-I}) \qquad \frac{\Gamma \vdash_\Sigma e : \Sigma_{x:\tau_1}\tau_2}{\Gamma \vdash_\Sigma \texttt{fst}(e) : \tau_1}(\Sigma\text{-E}_1)$$

$$\frac{\Gamma \vdash_\Sigma e : \Sigma_{x:\tau_1}\tau_2}{\Gamma \vdash_\Sigma \texttt{snd}(e) : [\texttt{fst}(e)/x]\tau_2}(\Sigma\text{-E}_2)$$

Definition: $\Gamma \vdash_\Sigma \tau_1 \updownarrow \tau_2 = \exists \tau_p\ .\ \epsilon|\Gamma \vdash_\Sigma \tau_1 <: \tau_p$ and $\epsilon|\Gamma \vdash_\Sigma \tau_2 <: \tau_p$.

Figure 4 Core Language Static Semantics.

One of the more interesting checks comes in the Match typing rule. Here, the tagged expression e_1 should be comparable to the tag n. If e_1 is tagged with n', then n' should be in the same branch of the tag hierarchy as n. This prevents pointless attempts to cast a tagged value to something of an entirely different tag "tree". Formally, this means that $\texttt{tagged}\ n$ and $\texttt{tagged}\ n'$ should have a common supertag in the hierarchy: there exists some n'' such that $\texttt{tagged}\ n <: \texttt{tagged}\ n''$ and $\texttt{tagged}\ n' <: \texttt{tagged}\ n''$. We define the $\updownarrow$ notation at the bottom of Figure 4 that specifies the need to go "up and down" the tag hierarchy.

Extracting a value from a tagged expression only requires that e is of type $\texttt{tagged}\ n$ in $\texttt{extract}(e)$. If e is tagged with n, then there is no harm in unwrapping e and typing it with the type that n tags.

As mentioned before, dependent sums are in the core language. The translation from the source to core language–shown in the next section–heavily uses dependent sums to convert classes, essentially because the type system must associate the tag generated for a class with the result type of the class constructor, which is also generated for that same class. We use the type abbreviation $\tau_1 \times \tau_2$ as a special case of the dependent sum $\Sigma_{x:\tau_1}.\tau_2$, where τ_2 does not use x. Furthermore, we can use the type abbreviation $\tau_1 \to \tau_2$ as a special case of the dependent function $\Pi_{x:\tau_1}.\tau_2$, where τ_2 does not depend on x.

As in typical dependent type systems, the type $\texttt{tagged}\ B$ is only well formed if B is in the context Γ. This leads to the avoidance problem [15] common in module systems, and requires the type system to use the subsumption rule when a variable goes of scope. For

$$S ::= \epsilon \mid S, c \rightsquigarrow p \qquad p ::= \varepsilon \mid c \rightsquigarrow p \qquad \frac{}{c \in c \rightsquigarrow p} \qquad \frac{c \in p \qquad c \neq c'}{c \in c' \rightsquigarrow p}$$

Notation: $S, x = S, x \rightsquigarrow \varepsilon$

Figure 5 Hierarchical Store Definition and Rules.

example, consider the following:

$\texttt{let } A : \texttt{int tag} = \texttt{newtag[int] in}$
$\texttt{let } x : \texttt{tagged } A =$
$\quad \texttt{let } B : \texttt{int tag extends } A = \texttt{subtag[int]}(A) \texttt{ in}$
$\quad \texttt{let } f : \Pi_{C:\texttt{int tag}}.\texttt{tagged } C = \lambda C{:}\texttt{int tag} \;.\; \texttt{new}(C; 1) \texttt{ in}$
$\quad \texttt{let } y : \texttt{tagged } B = f(B)$
$\quad \texttt{in } y$
...

Here f is a dependent function type, so when we invoke $f(B)$ we know that the result (bound to y) has type `tagged` B. On the other hand, if we then bind y to x in a scope where B is not visible (e.g. the second line of code above), we have to treat x as having the supertype `tagged` A because we must avoid mentioning B in the type.

3.3 Dynamics

The dynamics of the core language utilizes a hierarchical store (Figure 5) that represents all tags that have been generated in the execution of the program so far. Using a store to dynamically track tags is necessary because if a tag is created by a function, the function may be called an arbitrary number of times and thus generate an arbitrary number of tags. In the hierarchical store, a hierarchy is represented by a set of paths, each starting from a tag c and pointing to a (possibly empty) sub-path containing that tag's supertags, written as $c \rightsquigarrow p$. The empty path is ε and the end of a path $c \rightsquigarrow \varepsilon$ is shortened to c for notational clarity.

Operational semantics in the core language are expressed by small-step rules and value judgments. Value judgments are of the form

$$S \mid e \texttt{ val}$$

which states that e is a value in the operational semantics under the context of the hierarchical runtime store S. The transition relations are of the form

$$S \mid e \mapsto S' \mid e'$$

which states that e transitions to e' while potentially extending the store S to some S'. In the case the store is not modified, $S = S'$.

The expressions `newtag` and `subtag` create global tags into the hierarchical store. Top-level tags, which are of the form c, are created through evaluation of `newtag`. Note that `newtag` itself is not a value, but instead will evaluate to the fresh tag c.

This expresses the dynamic nature of tag creation in the core language. While tags were represented by local variables in the statics and syntax, the operational semantics

$$\frac{S \mid e \ \texttt{val}}{S \mid \texttt{new}(n; e) \ \texttt{val}}(\textsc{New-V}) \qquad \frac{}{S \mid \lambda x\colon \tau.e \ \texttt{val}}(\textsc{Lam-V})$$

$$\frac{c \in S}{S \mid c \ \texttt{val}}(\textsc{C-V}) \qquad \frac{\text{for each } i \quad S \mid e_i \ \texttt{val}}{S \mid \{\overline{f_i = e_i}\} \ \texttt{val}}(\textsc{Rec-V})$$

$$\frac{S \mid e_1 \ \texttt{val} \qquad S \mid e_2 \ \texttt{val}}{S \mid \langle e_1, e_2 \rangle \ \texttt{val}}(\textsc{Prod-V}) \qquad \frac{}{S \mid \langle\rangle \ \texttt{val}}(\textsc{Unit-V})$$

Figure 6 Core Language Value Judgments.

generate fresh global tags at runtime and store them in a hierarchy separate from the type hierarchy. Nonetheless, there is a direct relationship between the hierarchy created during static checking and the hierarchy created at runtime. Any tag n that extends an n' in the static tag hierarchy should have a corresponding c that contains c' (corresponding to n') in its path in the runtime hierarchical store.

Therefore `subtag` also creates a global tag c' and stores it into the hierarchical store, keeping track of the transitive tag path $c' \rightsquigarrow (c \rightsquigarrow p)$ that goes from c' to its parent tag c and on to the root of the hierarchy via the (possibly empty) path p. This leads to the store having the shape of an inverted tree, where each node points to its parent in the store. To check if a tag c is a subtag of another tag c', the dynamics will just follow the path from c back to the root, searching for c' along the way. The dynamic semantics express this search in the rules for $c \in p$.

The $\texttt{match}(e_1; c'; y.e_2; e_3)$ expression traverses the hierarchy to check if a tag is a valid subtag of the provided tag. Given a tagged value $e_1 = \texttt{new}(c; e)$, the dynamics of `match` check if c' is a supertag of c by traversing the path $c \rightsquigarrow p$ and searching for c'. This process is expressed by the relationship $c' \in c \rightsquigarrow p$ defined in Figure 5. A search fails when the path being searched is empty i.e. when $p = \varepsilon$. Upon a successful search, e is substituted for y in e_2. A failed search leads to e_3 instead.

Evaluating `extract` is straightforward. The dynamics do not check whether or not the tag given to `extract` is the same tag in side the `new` expression. Rather, evaluation simply strips off the tag from $\texttt{new}(c; e)$, leaving the untagged expression e. The typing rule for `extract`, together with the constraint that subtags must wrap a subtype of the type their supertags wrap, ensures that the extracted value is a subtype of the expected type.

The value judgments are formalized in Figure 6. Note that the dynamics of the core language use a call-by-value evaluation strategy. Consider the following partial expression:

$\texttt{let}\ x = \texttt{newtag}[\texttt{int}]\ \texttt{in}\ e$

As the `newtag` is evaluated eagerly, only one tag is generated in the global store, since the expression first steps to

$\texttt{let}\ x = c\ \texttt{in}\ e$

where c is a fresh tag in the global hierarchical store. If instead a call-by-name strategy was adopted, then the expression would step to

$[\texttt{newtag}[\texttt{int}]/x]e$

$$\frac{c \notin S}{S \mid \texttt{newtag}[\tau] \mapsto S, c \mid c} (\mapsto \text{CLS})$$

$$\frac{c' \notin S}{S, c \rightsquigarrow p \mid \texttt{subtag}[\tau](c) \mapsto S, c' \rightsquigarrow (c \rightsquigarrow p), c \rightsquigarrow p \mid c'} (\mapsto \text{CCLS})$$

$$\frac{S \mid e \mapsto S' \mid e'}{S \mid \texttt{new}(n; e) \mapsto S' \mid \texttt{new}(n; e')} (\mapsto \text{NEW})$$

$$\frac{S \mid e \mapsto S' \mid e'}{S \mid \texttt{match}(e; c; y.e_2; e_3) \mapsto S' \mid \texttt{match}(e'; c; y.e_2; e_3)} (\mapsto \text{MATCH})$$

$$\frac{S, c \rightsquigarrow p \mid e \ \texttt{val} \qquad c' \in c \rightsquigarrow p}{S, c \rightsquigarrow p \mid \texttt{match}(\texttt{new}(c; e); c'; y.e_2; e_3) \mapsto S, c \rightsquigarrow p \mid [\texttt{new}(c; e)/y]e_2} (\mapsto \text{MATCHSUC})$$

$$\frac{S, c \rightsquigarrow p \mid e \ \texttt{val} \qquad c' \notin c \rightsquigarrow p}{S, c \rightsquigarrow p \mid \texttt{match}(\texttt{new}(c; e); c'; y.e_2; e_3) \mapsto S, c \rightsquigarrow p \mid e_3} (\mapsto \text{MATCHFAIL})$$

$$\frac{S \mid e \mapsto S' \mid e'}{S \mid \texttt{extract}(e) \mapsto S' \mid \texttt{extract}(e')} (\mapsto \text{UNTAG1})$$

$$\frac{S \mid e \ \texttt{val}}{S \mid \texttt{extract}(\texttt{new}(_; e)) \mapsto S \mid e} (\mapsto \text{UNTAG2})$$

Figure 7 Core Language Dynamic Semantics.

which would create a tag for every binding site of x in e – and the tags used to tag values would be different from the tags used as matching targets, causing all matches to fail! Thus the choice of call-by-value is essential.

Evaluation semantics for expressions related to hierarchical tagging can be found in Figure 7. Congruence rules for records, tuples, and let-bindings have been omitted for the sake of brevity.

3.4 Type Soundness

Stated here are the basic type safety theorems for the core language.

Preservation. If $\Gamma \vdash_\Sigma e : \tau$, and $S \mid e \mapsto S' \mid e'$, then $\exists \Sigma' \supseteq \Sigma$ such that $\Gamma \vdash_{\Sigma'} e' : \tau$.
Progress. If $\epsilon \vdash_\Sigma e : \tau$ and $\Sigma \vdash S$, then either $S \mid e \ \texttt{val}$ or $S \mid e \mapsto S' \mid e'$.

The proofs are sketched in the supplementary material [20].

4 Source Language Translation

We would like to define the semantics of our source language. One option is to do so directly, via typing rules and an operational semantics. However, the source language includes a lot of inessential complexity, mainly having to do with the built-in features of classes and methods, all of which are well understood. A source-level semantics would therefore include a fair amount of complexity, but it would be complexity that is not very interesting. Furthermore,

we cannot build directly on previous formal systems, because we are not aware of prior work that includes first-class classes and tag tests in a statically typed setting.

For these reasons–and also considering space constraints–we follow Harper and Stone [16] in defining the semantics of the source language via a translation in to the core language defined in section 3. Our translation has two judgments. Type translation is of the form:

$$\Gamma \vdash \tau \Rightarrow \tau^\dagger$$

where τ is a type in the source language and $\tau^\dagger$ is a type in the core language. Expression translation is of the form

$$\Gamma \vdash e \Rightarrow e^\dagger : \tau^\dagger$$

where e is an expression in the source language and $e^\dagger$ along with $\tau^\dagger$ are respectively an expression and its type in the core language. The translation is primarily concerned with converting classes and objects to the core language that does not have such expressions.

Our translation is based on Bruce et al.'s encoding [5]. In this approach, an object is encoded as a record through the use of recursive types. This encoding is commonly used when translating expressions in an object calculus into a typed lambda calculus. However, the encoding in Bruce et al. leaves out the ability to perform instance checks on an object at run time, as a naive recursive record translation discards the relation between the object and the class it instantiates. If there is no tag involved, then it is impossible to create an object and use `match` to check if it is an instance of a class.

Our solution to the problem is to wrap the encoded object in a `tagged` n, where n is a tag representing a class in the source language. Then match statements in the source language can be translated directly into matches against the corresponding tag in the core language. On the other hand, to invoke a method or access a field, our translation will need to first use `extract` to get a record from the tagged value, then select the appropriate method or field and invoke or read it.

The translation of classes is more complicated. A class must encapsulate both the tag that tags its instances, and a constructor that is used to create instances. We therefore translate classes into a pair consisting of a tag and a constructor function that takes initial values for the fields and generates a tagged value. Because the type of the constructor depends on the value of the tag, we do not use an ordinary pair but rather a dependent sum type. Finally, methods and fields within the class can refer to the class itself, so we wrap the pair in a recursive type.

We can now understand the type translation rules in Figure 8. An object type is translated by looking up the type of the class, recursively translating the class type into a recursive dependent pair type representing the class, and then generating a core type `tagged fst`$($`unfold`$(x))$, a value tagged with the first element of the unfolded recursive pair. The rule for translating function types is simple: it just recursively translates the types of the function argument and result.

The translation for class types begins in the premises by recursively translating the field and method types. When doing so, the type x being defined must be in scope, because the methods and fields might use that type in their definitions (e.g. when defining a binary method or a recursive data structure). We generate a dependent pair, naming the first element of the pair x_{tag} so we can use it in the second element. The first element is a tag that extends the tag of x' – which will be `fst`(x') in the translation – if a parent class x' was specified. Note that we use [] for optional syntax in the rule – all brackets must be present or absent when the rule is instantiated. We abbreviate the class body as $I(x)$ as the class

Let $I(x)$ be the type interface a given class defines, i.e. $I(x) = \{\overline{f : \tau_f}, \overline{m : \tau_m}\}$ for $\texttt{class}\ x\ I(x)\ \texttt{in}\ e$, and similarly for subclasses. Then let $I(\tau') = \{\overline{f : [\tau'/x]\tau_f}, \overline{m : [\tau'/x]\tau_m}\}$ to represent changing all instances of x in the class interface. We abbreviate $I^\dagger(x) = \{\overline{f : \tau_f^\dagger}, \overline{m : \tau_m^\dagger}\}$.

Type Translation $\Gamma \vdash \tau \Rightarrow \tau^\dagger$

$$\frac{\Gamma(x) = \tau \qquad \Gamma \vdash \tau \Rightarrow \mu t.\Sigma_{x_{\texttt{tag}}:\ldots}(\overline{\tau_f^\dagger} \rightarrow \texttt{tagged}\ x_{\texttt{tag}})}{\Gamma \vdash x\ \texttt{obj} \Rightarrow \texttt{tagged}\ \texttt{fst}(\texttt{unfold}(x))}$$

$$\frac{\Gamma \vdash \tau_1 \Rightarrow \tau_1^\dagger \qquad \Gamma \vdash \tau_2 \Rightarrow \tau_2^\dagger}{\Gamma \vdash \tau_1 \rightarrow \tau_2 \Rightarrow \tau_1^\dagger \rightarrow \tau_2^\dagger}$$

$$\frac{\begin{array}{ll}\forall f_i : \tau_{f_i} & \Gamma, x : \texttt{class}\ \ x\ I(x)[\texttt{extends}\ \ x'] \vdash \tau_{f_i} \Rightarrow \tau_{f_i}^\dagger \\ \forall m_i : \tau_{m_i} & \Gamma, x : \texttt{class}\ \ x\ I(x)[\texttt{extends}\ \ x'] \vdash \tau_{m_i} \Rightarrow \tau_{m_i}^\dagger\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{class}\ \ x\ I(x)[\texttt{extends}\ \ x'] \Rightarrow \\ \mu t.\Sigma_{x_{\texttt{tag}}:(I_\tau^\dagger(t)\ \texttt{tag}\ [\texttt{extends}\ \texttt{fst}(\texttt{unfold}(x'))])}(\overline{\tau_f^\dagger} \rightarrow \texttt{tagged}\ x_{\texttt{tag}})\end{array}}$$

Figure 8 Source to Core Type Translation.

name x is recursively bound in the body; the notation $I^\dagger(x)$ defined at the top of the figure simply applies the type translation recursively to the elements of the class body. The class body then becomes a record type.

The second element of the dependent pair represents the constructor, which takes as arguments the types of the class's fields, and returns the tagged type. We wrap the entire pair in a recursive type so that the class type is properly bound in the class body.

We turn now to the expression translation rules in Figures 9 and 10. For the sake of brevity, we omit the translation rules for lambdas, function application, and let-bindings; these rules simply translate the component types and expressions recursively. The first rule translates the `new` expression, first by translating the subexpressions, looking up the translated type of the class being instantiated, and invoking the second member of the unfolded pair representing the class (i.e. the constructor function) with the translated arguments. The result is tagged with the class's tag.

The second rule is for method calls. It translates the receiver. Because the receiver must have been an object type in the source language, we know its type will be of the form $\texttt{tagged}\ \texttt{fst}(\texttt{unfold}(x))$; looking up the type of x in the context, we find the underlying record type $I_\tau^\dagger(\tau^\dagger)$, and select the type of the method m. The rule for field reads on the top right is analogous. The rule for match, second from the top on the right, translates all the component parts and creates a core-level `match` statement that extracts the tag from the first component of the unfolded class value.

Finally, the two rules at the bottom translate class expressions. The result of translation is a recursive value, which we implement with a `letrec` and a `fold`. We construct the pair with a new tag holding the translated record type, and a constructor function that uses another `letrec` to bind `this` to a tagged value that wraps the record itself. The variant on the right is identical to that on the left, except that a subtag is created.

Let $I(x)$ be the type interface a given class defines, i.e. $I(x) = \{\overline{f : \tau_f}, \overline{m : \tau_m}\}$ for $\texttt{class}\ x\ I(x)\ \texttt{in}\ e$, and similarly for subclasses. Then let $I(\tau') = \{\overline{f : [\tau'/x]\tau_f}, \overline{m : [\tau'/x]\tau_m}\}$ to represent changing all instances of x in the class interface. We abbreviate $I^\dagger(x) = \{\overline{f : \tau_f^\dagger}, \overline{m : \tau_m^\dagger}\}$. Finally, $I(\tau)[f]$ yields the type associated with field f in the interface $I(\tau)$.

Expression Translation $\Gamma \vdash e \Rightarrow e^\dagger : \tau^\dagger$

$$\frac{\forall e_i \in \overline{e} \quad \Gamma \vdash e_i \Rightarrow e_i^\dagger : \tau_i^\dagger \qquad \Gamma(x) = \tau \qquad \Gamma \vdash \tau \Rightarrow \mu t.\Sigma \ldots}{\Gamma \vdash \texttt{new}(x; \overline{e}) \Rightarrow (\texttt{snd}(\texttt{unfold}(x)))(\overline{e^\dagger}) : \texttt{tagged}\ \texttt{fst}(\texttt{unfold}(x))}$$

$$\frac{\begin{array}{l}\Gamma \vdash e \Rightarrow e^\dagger : \texttt{tagged}\ \texttt{fst}(\texttt{unfold}(x)) \qquad \Gamma(x) = \tau \\ \Gamma \vdash \tau \Rightarrow \tau^\dagger \qquad \tau^\dagger = \mu t.\Sigma_{x_{\texttt{tag}} : (I_\tau^\dagger(t)\ \texttt{tag}\ \ldots)} \cdots\end{array}}{\Gamma \vdash e.m \Rightarrow \texttt{extract}(e^\dagger).m : I_\tau^\dagger(\tau^\dagger)[m]}$$

$$\frac{\begin{array}{l}\forall m_i : \tau_{m_i} = e_{m_i} \qquad \Gamma, x : \texttt{class}\ x\ I(x) \vdash e_{m_i} \Rightarrow e_{m_i}^\dagger : \tau_{m_i}^\dagger \\ \Gamma \vdash \texttt{class}\ x\ I(x) \Rightarrow \tau_x^\dagger \qquad \tau_x^\dagger = \mu t.\Sigma_{x_{\texttt{tag}} : (I_\tau^\dagger(t)\ \texttt{tag})} \cdots \\ \Gamma, x : \texttt{class}\ x\ I(x) \vdash e \Rightarrow e^\dagger : \tau^\dagger\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{class}\ x\ \{\overline{f : \tau_f}, \overline{m : \tau_m = e_m}\}\ \texttt{in}\ e \Rightarrow \\ \texttt{letrec}\ x = \texttt{fold}[\tau_x^\dagger](\ \langle \texttt{newtag}[I_\tau^\dagger(t)],\ \ \lambda \overline{y} : \overline{\tau_f^\dagger}. \\ \qquad \texttt{letrec this} =\ \ \texttt{let}\ o = \{\overline{f = y}, \overline{m = e_m^\dagger}\}\ \ \texttt{in new}(\texttt{fst}(\texttt{unfold}(x)); o) \\ \qquad \texttt{in this}\rangle)\ \ \texttt{in}\ e^\dagger\ \ : \tau^\dagger\end{array}}$$

Figure 9 Source Language to Core Language Expression Translation (Part 1 of 2).

Properties. Note that the translation given above completely defines the semantics for the source language, following the style of semantics given by Harper and Stone for Standard ML [16]. For example, typechecking a source program succeeds exactly if the translation rules apply and the result of translation is well-typed in the target language. This means that there is no separate type safety theorem to prove for the source language; the source language is type safe because it is defined by translation into a type safe target language.

Discussion. Our translation motivates an interesting feature of our core language: the separation of `extract` from `match`. A more obvious choice would have been to combine these, providing a $\texttt{match}(e; n; x.e; e)$ statement that binds x to the extracted value when the match succeeds. This choice works well for *non*-hierarchical tags, and indeed is the construct used in conventional sum types as well as ML's `exn` type and Harper's classified types [14]. It does not work for *hierarchical* tags, however. To see why, imagine we have a three-level hierarchy, with C a subtag of B and B a subtag of A. We can create a `tagged` C and treat it, by subsumption, as a `tagged` A. If we match against B, we do not want to immediately extract the contents, because we want to not forget that it is really a C in case we try to

$$\frac{\Gamma \vdash e \Rightarrow e^\dagger : \texttt{tagged}\ \texttt{fst}(\texttt{unfold}(x)) \qquad \Gamma(x) = \tau \qquad \Gamma \vdash \tau \Rightarrow \tau^\dagger \qquad \tau^\dagger = \mu t.\Sigma_{x_{\texttt{tag}}:(I^\dagger_\tau(t)\ \texttt{tag}\ \ldots)} \cdots}{\Gamma \vdash e.f \Rightarrow \texttt{extract}(e^\dagger).f : I^\dagger_\tau(\tau^\dagger)[f]}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1^\dagger : \tau_1^\dagger \qquad \Gamma \vdash e_3 \Rightarrow e_3^\dagger : \tau^\dagger \qquad \Gamma, y : x\ \texttt{obj} \vdash e_2 \Rightarrow e_2^\dagger : \tau^\dagger}{\Gamma \vdash \texttt{match}(e_1; x; y.e_2; e_3) \Rightarrow \texttt{match}(e_1^\dagger; \texttt{fst}(\texttt{unfold}(x)); z.[z/y]e_2^\dagger; e_3^\dagger)\ :\ \tau^\dagger}$$

$$\frac{\forall m_i : \tau_{m_i} = e_{m_i} \qquad \Gamma, x : \texttt{class}\ \ x\ I(x)\ \ \texttt{extends}\ \ x' \vdash e_{m_i} \Rightarrow e^\dagger_{m_i} : \tau^\dagger_{m_i} \qquad \Gamma \vdash \texttt{class}\ \ x\ I(x)\ \ \texttt{extends}\ \ x' \Rightarrow \tau^\dagger_x \qquad \tau^\dagger_x = \mu t.\Sigma_{x_{\texttt{tag}}:(I^\dagger_\tau(t)\ \texttt{tag}\ \ldots)} \cdots \qquad \Gamma, x : \texttt{class}\ \ x\ I(x)\ \ \texttt{extends}\ \ x' \vdash e \Rightarrow e^\dagger : \tau^\dagger}{\begin{array}{l}\Gamma \vdash \texttt{class}\ \ x\ \{\overline{f : \tau_f}, \overline{m : \tau_m = e_m}\}\ \ \texttt{extends}\ \ x'\ \ \texttt{in}\ \ e \Rightarrow \\ \texttt{letrec}\ \ x = \texttt{fold}[\tau^\dagger_x](\ \ \langle \texttt{subtag}[I^\dagger_\tau(t)](\texttt{fst}(\texttt{unfold}(x'))),\quad \lambda\overline{y} : \overline{\tau^\dagger_f}. \\ \qquad \texttt{letrec this} =\ \ \texttt{let}\ \ o = \{\overline{f = y}, \overline{m = e^\dagger_m}\}\ \ \texttt{in}\ \texttt{new}(\texttt{fst}(\texttt{unfold}(x)); o) \\ \qquad \texttt{in this}\rangle)\ \ \texttt{in}\ \ e^\dagger\ \ : \tau^\dagger\end{array}}$$

Figure 10 Source Language to Core Language Expression Translation (Part 2 of 2).

match it further later on. Implementing hierarchical tags as nested tagging also fails, because then `tagged` C is not a subtype of `tagged` A.

This insight was not obvious to us when we began; our first, failed, translation attempted to use a single construct combining `extract` and `match`. We believe it will be useful both to theorists who will further study the type-theoretic foundations of objects, and to language designers who wish to provide primitives sufficient to encode rich object systems.

5 Implementation

We implemented the core functionality of tags as a contribution to the open source Wyvern programming language [24] developed at Carnegie Mellon University (CMU) in the USA and Victoria University of Wellington in New Zealand, although the implementation of certain supporting features such as dependent function types are not yet complete. The Wyvern interpreter's open source implementation is available [5] and this paper is accompanied by an artifact that was accepted and archived. The syntax of Wyvern's tag support differs slightly from the source-level language presented in Section 2 to better harmonize with the other features of Wyvern, but the underlying concepts remain the same and the implementation is informed by the theory presented here.

[5] `https://github.com/wyvernlang` (our contribution is in the TaggedTypes branch)

```
1  tagged type Window
2    def draw():Str
3
4  type WindowMod
5    tagged class Win [case of Window]
6      class def make():Win = new
7      def draw():Str = ""
8
9  val basicWindow:WindowMod = new
10   tagged class Win [case of Window]
11     class def make():Win = new
12     def draw():Str = "blank␣window"
13
14 def makeBordered(wm: WindowMod):WindowMod = new
15   tagged class Win [case of wm.Win]
16     class def make():Win = new
17     def draw():Str = "bordered␣window"
18
19 def makeScrollable(wm: WindowMod):WindowMod = new
20   tagged class Win [case of wm.Win]
21     class def make():Win = new
22     def draw():Str = "scrollable␣window"
23
24 def userWantsBorder():Bool = true
25
26 val winMod:WindowMod =
27   if (userWantsBorder())
28   then
29       makeBordered(basicWindow)
30   else
31       basicWindow
32
33 val bigWinMod:WindowMod = makeScrollable(winMod)
34 val smallWin = winMod.Win.make()
35 val bigWin = bigWinMod.Win.make()
36
37 def screenCap(w:Window):Str
38   match(w):
39     bigWinMod.Win => "big"
40     default => "small"
41
42 val result = screenCap(bigWin)
43
44 result // result == "big"
```

Figure 11 Wyvern Example with Tagged Types.

Bordered Window Example. To understand our implementation of tags in Wyvern, consider the Wyvern version of the bordered window example (Section 2.1), shown in Figure 11. The code is one of the unit tests, and typechecks and executes correctly in the current Wyvern implementation.

Before we implemented tags, Wyvern supported `type` and `class` declarations. In our tags extension, both of them can be declared as `tagged`. If the type or class declaration is nested within a function – e.g, the `tagged` classes declared in the `makeBordered` and `makeScrollable` – then a fresh tag is created every time the function is executed, as in the semantics of the source and core languages we defined.

Wyvern does not have a way of specifying a class type directly, but we can specify the type of an object that has a class nested within it – that object plays the role of a module. For example, `WindowMod` is the type of a module that defines a `Win` class that (transitively) extends `Window`. The Wyvern syntax for specifying a sub-tagging relationship is `case of`, which is analogous to `extends` in Java.

Here is how `makeBordered` defines a mixin: it accepts a `WindowMod` module as an argument and produces another `WindowMod` module, where the class nested in the result has a subtag of the class nested in the argument. The `makeScrollable` function is similar. The `winMod` module decides whether to apply `makeBordered` dynamically based on the result of `userWantsBorder`, demonstrating Wyvern's support for dynamic composition. In this test case, `userWantsBorder` always returns `true`, but Wyvern's interpreter doesn't know that until it executes the function.

The `screenCap` function shows how even if we do not statically know the type of `w`, we can dynamically match against a tagged class defined in any module, such as `bigWinMod`. When calling `screenCap` with `bigWin` as argument, the match succeeds.

Implementation. Wyvern is currently implemented in an interpreter; when run on a piece of source code, the source is parsed, typechecked, and then interpreted. The operation of the typechecker and interpreter roughly follows the semantics outlined in this paper, appropriately adapted to the more practical design of the Wyvern language. For example, each time we create a new object which contains a nested tagged type, we create a new object in the interpreter representing a fresh tag, and we associate it with the nested type. Objects created of that type are then associated with that tag, and the tag can be checked when `match` statements are executed.

The design of Wyvern includes not only extensible tags as described in this paper, but also closed tagged unions. A `tagged` type can be declared with the `[comprises` $T_1, T_2, ...T_n$`]` modifier, which specifies that the type being declared has only the subtags listed in the `comprises` clause. The listed types must declare themselves as a case-of of the parent tag, and the typechecker ensures that no other types are declared to be a case of that parent tag. This allows tagged types to simulate not only objects, but also algebraic datatypes.

6 Modeling Multiple Inheritance

So far, we have developed a foundational type theory that can explain the common constructs of single-inheritance object-oriented languages. OO languages with multiple inheritance, however, cannot be modeled with our calculus so far. A key barrier is subtyping: we want the type `tagged` T to be a subtype of the type `tagged` T_i for each supertag T_i of T, but this cannot be achieved in our current system. This is because in the system presented so far, each tag can have only one parent, and each object can have only one (outermost) tag. Multiple tagging (e.g. by m and n) can be achieved by nesting tags (e.g. a type `tagged` n, where n's inner type is `tagged` m), but this does not provide the desired subtyping properties (in this case, `tagged` n is not a subtype of `tagged` m). The issue is that a tagged object is not identical to its contents. One could imagine trying to make a tagged object semantically identical to its contents, but this seemed both awkward to us and inconsistent with prior type-theoretic models of tags (e.g. in Glew's work and standard ML [13, 14, 16]) and we did not pursue it.

While the restriction to single inheritance is useful to keep the system simple in the main presentation above, it can easily be relaxed. Figure 12 shows the changes to the syntax and static semantics necessary to support OO languages with multiple inheritance. We allow a subtag to extend multiple previously-defined tags; the type being tagged must be a subtype of all of the types tagged by its supertags. When creating an object, it can be tagged with multiple tags as well, and the initial value provided must likewise have a type

e ::= $\texttt{newtag}[\tau]$ | $\texttt{subtag}[\tau](n)$ | $\texttt{new}(N; e)$ | $\texttt{match}(e; N; x.e; e)$ | $\texttt{extract}(n; e)$ | ...

τ ::= $Tag(\tau)$ | $\texttt{tagged}\ N$ | ...

$Tag(\tau)$ ::= $\tau\ \texttt{tag}$ | $\tau\ \texttt{tag extends}\ n$

Where $N = \{\overline{n}\}$.

$$\frac{\forall n \in N \qquad \Gamma \vdash_\Sigma n : Tag(\tau) \qquad \Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \texttt{new}(N; e) : \texttt{tagged}\ N}\text{(MTag)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma e_1 : \texttt{tagged}\ N' \\ \Gamma \vdash_\Sigma \texttt{tagged}\ N \subseteq \texttt{tagged}\ N' \qquad \Gamma, x : \texttt{tagged}\ N \vdash_\Sigma e_2 : \tau \qquad \Gamma \vdash_\Sigma e_3 : \tau\end{array}}{\Gamma \vdash_\Sigma \texttt{match}(e_1; N; x.e_2; e_3) : \tau}\text{(MMatch)}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{tagged}\ N \qquad n \in N \qquad \Gamma \vdash_\Sigma n : Tag(\tau)}{\Gamma \vdash_\Sigma \texttt{extract}(n; e) : \tau}\text{(MExtract)}$$

Where $\Gamma \vdash_\Sigma \texttt{tagged}\ N \subseteq \texttt{tagged}\ N'$ iff $\forall n \in N. \exists n' \in N'. \Gamma \vdash_\Sigma \texttt{tagged}\ n \updownarrow \texttt{tagged}\ n'$

Figure 12 Static Semantics for Multiple Tags.

that matches (perhaps via subtyping) the types expected by each tag.[6] Finally, `match` can compare against a set of tags. We leave `extract` to get the contents for one tag, as we would otherwise need to compute an intersection type – something that is feasible but would unnecessarily complicate our presentation. The extensions to the subtyping judgments, value judgments, and dynamic semantics are straightforward; the latter two figures are left to the supplementary material [20] for space reasons.

7 Related Work

First-class classes have been used in Racket along with mixin support; [9] describes the dynamically-typed design, which supports an `implementation?` operation that provides instance (or tag) checking. The topic of first-class classes with static typing has been explored by Takikawa, et al. [27]. They design a gradual typing system to support statically-typed and dynamically-typed portions of their language. They also demonstrate the ability to use mixins via row polymorphism for classes. However, the use of row polymorphism gives their type system a structural flavor; it does not express the concept of an object that is associated with a particular tag.

Reppy and Riecke extend Standard ML with objects and generalize pattern matching to typecase [26]. The extension, called Object ML, adds objects, subtyping, and heterogeneous collections to SML. The design of objects in OML is motivated by recursive types, but opts to keep objects as second-class declarations. Furthermore, they also use SML structs to

[6] We debated between supporting multiple inheritance with tags that have multiple parents vs. objects that have multiple tags. We chose the multiply-tagged object solution as it seemed cleaner and easier to formalize; for example, we can leave the formalization of the tag hierarchy unchanged. Furthermore, the multiply-tagged object design matches the intuition that typical uses of multiple inheritance can be viewed a combination of several dimensions of reuse, each of which can be described in a single inheritance hierarchy [22]. Nevertheless, both choices are possible.

encode classes in OML. As such, their tags are second-class, while our source language opts to use first-class classes and thus translates to first-class tags. Other work on extensible datatypes in functional languages also incorporates second-class tags, thus differing from our results in similar ways [3, 23].

Abadi, et al introduced a means of encoding dynamic types in a statically-typed language through the use of (non-hierarchical) tagging and typecasing [1]. Vytiniotis, et al. explores open and closed type casing through their $\lambda_{\mathcal{L}}$ language [28]. They were primarily concerned with open and closed datatypes and those two different forms of ad-hoc polymorphism. They analyze sets of tags in order to statically check whether or not a given typecase type checks. They do not provide typecasing with subtyping or any form of hierarchical typing.

Our core language is similar to Glew's source language, which was used to motivate a more general use of hierarchical store for modeling type dispatch [13]. However, Glew's language does not use the static type system to keep track of relationships between tags, opting instead to rely on the operational semantics to uphold the theorems of type safety. In Glew's system, for example, the static type of an object is simply `tagged`; it does not track the object's class (or even a superclass), which is a basic capability of OO type systems that we wish to model.

Others have explored tag-like constructs in the context of object-oriented languages. The Unity language defined a *brand* construct that provided nominal types in an otherwise structurally-typed language; at run time, brands are associated with tags that are then used for dispatch [21]. Brands in unity are second-class. Concurrent with our work, Jones *et al.* defined a first-class brand system for Grace [19]. Their focus is on a primitive brand construct that adds nominality to an otherwise structurally-typed system; reflection is used to test brand membership. In contrast, our focus is on the type theory of tags, and thus our work differs in providing primitive operations taken from type theory, including an explicit match operation, and statically tracking the sub-tagging hierarchy.

8 Conclusion

The previous sections of this paper explored a foundational account of class-based object-oriented languages: one that supports dynamic class creation and composition out of mixins, and explains classes in terms of a novel primitive hierarchical tag construct inspired by the type theory of extensible sums. We hope that this account contributes to a better understanding of the relationship between the constructs of typical object-oriented programming languages and the fundamental elements of type theory. Our account highlights that the most simple tag constructs are insufficient to model objects in a type-preserving way, yet shows that small extensions to support static reasoning about tag hierarchy, to provide an appropriate `match` construct, and to statically track the tag associated with each object, are sufficient for modeling objects. We discovered an interesting subtlety, discussed at the end of section 4, in the need to separate `match` from `extract`, and in section 6 we discussed how our approach naturally generalizes to support objects with multiple tags and tags with multiple supertags. These results suggest that functional programming languages, such as Standard ML, that already provide an extensible tag mechanism [16] could gain expressiveness by adopting the enhanced constructs in our theory, perhaps in conjunction with mechanisms such as refinement types [11].

We expect the theory presented here to contribute to the design of Wyvern [24], a language that already supports dynamic class creation and composition, but may benefit from support for multiple tags as well. The flexibility of the theory suggests that it may enable statically

typed, nominal OO languages to express many useful design idioms that, at present, are limited to dynamically-typed or structurally-typed languages.

Acknowledgments. This work was supported by the U.S. National Security Agency lablet contract #H98230-14-C-0140. We acknowledge an invaluable input of Benjamin Chung – the primary maintainer of the Wyvern Compiler.

References

1. Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.
2. Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *European Conference on Object-Oriented Programming*, 1999.
3. François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *Principles of Programming Languages*, 1997.
4. Gilad Bracha and William Cook. Mixin-based inheritance. In *Object-Oriented Programmming, Systems, Languages, and Applications*, 1990.
5. Kim Bruce, Luca Cardelli, and Benjamin Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, 1999.
6. Luca Cardelli. Amber. In *Combinators and Functional Programming Languages*, 1986.
7. David Clarke, James Noble, and John Potter. Simple ownership types for object confinement. In *European Conference on Object-Oriented Programming*, 2001.
8. Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *European Conference on Object-Oriented Programming*, 2007.
9. Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems*, 2006.
10. Matthew Flatt, Shriram Krishnamurthi, and Mattias Felleisen. Classes and mixins. In *Principles of Programming Languages*, 1998.
11. Tim Freeman and Frank Pfenning. Refinement types for ML. In *Programming Language Design and Implementation*, June 1991.
12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
13. Neal Glew. Typed dispatch for named hierarchical types. In *International Conference on Functional Programming*, 1999.
14. Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2013.
15. Robert Harper and Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages (Edited by Benjamin C. Pierce)*, chapter Design Considerations for ML-Style Module Systems. MIT Press, 2005.
16. Robert Harper and Chris Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, Carnegie Mellon University, 1997.
17. Susumu Hayashi. Singleton, union and intersection types for program extraction. In *Theoretical Aspects of Computer Software*, 1991.
18. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.
19. Timothy Jones, Michael Homer, and James Noble. Brand objects for nominal typing. In *European Conference on Object-Oriented Programming*, 2015.

20 Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. A theory of tagged objects (with supplementary material). Technical Report 15-03, ECS, VUW, 2015. http://ecs.victoria.ac.nz/Main/TechnicalReportSeries.
21 Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *European Conference on Object-Oriented Programming*, 2008.
22 Donna Malayeri and Jonathan Aldrich. CZ: Multiple inheritance without diamonds. In *Object-Oriented Programmming, Systems, Languages, and Applications*, 2009.
23 Todd Millstein, Colin Bleckner, and Craig Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *Transactions on Programming Languages and Systems*, 26(5):836–889, 2004.
24 Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: A simple, typed, and pure object-oriented language. In *Mechanisms for Specialization, Generalization, and Inheritance (MASPEGHI)*, 2002.
25 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
26 John Reppy and Jon Riecke. Simple objects for Standard ML. In *Programming Language Design and Implementation*, 1996.
27 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Mattias Felleisen. Gradual typing for first-class classes. In *Object-Oriented Programmming, Systems, Languages, and Applications*, 2012.
28 Dimitrios Vytiniotis, Geoffrey Washburn, and Stephanie Weirich. An open and shut typecase. In *Programming Language Design and Implementation*, 2005.

Brand Objects for Nominal Typing

Timothy Jones, Michael Homer, and James Noble

Victoria University of Wellington
New Zealand
{tim,mwh,kjx}@ecs.vuw.ac.nz

Abstract

Combinations of structural and nominal object typing in systems such as Scala, Whiteoak, and Unity have focused on extending existing nominal, class-based systems with structural subtyping. The typical rules of nominal typing do not lend themselves to such an extension, resulting in major modifications. Adding object branding to an existing structural system integrates nominal and structural typing without excessively complicating the type system. We have implemented *brand objects* to explicitly type objects, using existing features of the structurally typed language Grace, along with a static type checker which treats the brands as nominal types. We demonstrate that the brands are useful in an existing implementation of Grace, and provide a formal model of the extension to the language.

1998 ACM Subject Classification D.3.3 Classes and objects

Keywords and phrases brands, types, structural, nominal, Grace

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.198

Supplementary Material ECOOP Artifact Evaluation approved artifact available at
http://dx.doi.org/10.4230/DARTS.1.1.4

1 Introduction

Most statically typed object-oriented languages use nominal subtyping. From Simula [5] and C++, through to Java, C♯ and Dart, an instance of one type can only be considered an instance of another type if the subtyping relationship is declared in advance, generally at the time the subtype is declared.

Some modern languages have adopted structural subtyping. In Go, for example, types are declared as interfaces, and an object conforms to a type if the object declares at least the methods required by the interface [41]. As well as Go's interfaces, Emerald is structurally typed [9], as is OCaml's object system [31] and Trellis/OWL [42]. Structural types have also been used to give types post-hoc to dynamically typed languages: Strongtalk originally supported structural types for Smalltalk [14], and Diamondback Ruby uses structural types for Ruby [23].

Given that nominal and structural typing both have advantages, there have been attempts to combine them both in a single language. The Whiteoak language [24] begins with Java's nominal type system and adds in support for structural types. Around the same time, Scala 2.6 [36] added structural types, again on top of a language with a nominal type system. The Unity language design similarly adds structural types onto a nominal class hierarchy [32].

The key argument of this paper is that adding nominal types to a structural type system requires a relatively smaller amount of effort. The corollary to this argument is that adding structural types into a nominal language – the direction of most existing approaches to the problem – does things backwards. This argument is based on our experience building a

© Timothy Jones, Michael Homer, and James Noble;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 198–221

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

branding mechanism in Grace, an object-oriented language with a standard structural type system [6, 7]. Both the static and dynamic nature of brands have been implemented using existing features of Grace, with minimal changes to the language.

We validate our argument by contributing:

- A practical design of *brand objects* for nominal typing on top of Grace's existing structural type system.
- An implementation of brand objects and a static nominal type checker in Hopper, a prototype implementation of Grace.
- Case studies of branding for various components of the language design in Hopper.
- A formal model of brands as nominal types as an extension to an existing model of a subset of the language.

The remainder of this paper proceeds as follows: Section 2 presents our motivation and design for branded types. Section 3 describes the implementation. Section 4 presents a series of case studies validating the utility of our branding system in the existing implementation. Section 5 formalizes our design, and proves soundness. Section 6 discusses alternative approaches, Section 7 discusses related work, and Section 8 concludes.

2 Brands

Nominal and structural typing both have advantages [32, 24]. Structural typing decouples an object's type – the set of methods to which it can respond – from the object's implementation (usually a class). Structural types can be declared at any time, in any part of the program, and still be relevant to any object with the appropriate interface. Any object that conforms to a structural type can be used wherever an instance of that structural type is required, even though the object's declaration did not declare that it implemented the type – among the reasons that Go adopted structural typing [41].

Being based solely on objects' interfaces rather than their implementations, structural types correspond to the conceptual model of object-oriented programming where individual objects communicate only via their interfaces, with their implementations encapsulated [19]. The clear separation between structural types and their implementing classes, and the ease of defining types independently from classes works well with gradual and pluggable typing [13], so programmers can begin by writing programs without types, and then add types later as the need increases.

On the other hand, nominal subtype relationships must be designed and declared by programmers, meaning they can capture programmers' intentions explicitly. Nominal subtyping can make finer distinctions between objects than structural subtyping: a structural system cannot distinguish between two different classes that have the same external interface, whereas a nominal system can distinguish between every implementation of every interface. Because nominal types can distinguish between different implementations (classes), compilers and virtual machines can optimize object allocation and method execution for particular implementations – for example, allocating machine integers and compiling arithmetic without any method dispatch.

As most languages are nominally typed, most of the major platforms for object-oriented languages (the Java Virtual Machine and the Common Language Runtime) are themselves nominally typed, so interoperability with VMs and languages is assisted by nominal typing. Pedagogically, nominal subtyping ensures every type has a name, so compilers and IDEs (especially their error messages) can refer to types by name, making teaching and debugging easier. Every nominal type has an explicit, unique, declaration in the program, a declaration that describes its relationships with all its supertypes, so class and type hierarchies can

be understood in a straightforward manner. These advantages are among the reasons that Strongtalk, for example, moved from structural to nominal typing [12, 14].

In order to implement brands, we have added *brand objects* to the Grace [6] programming language, extending its existing structural typing mechanism. A brand object represents a unique marker that can be applied to an object and subsequently detected, either statically or dynamically; the effects of these objects are similar to the branded types in Modula-3 [35]. We argue that, in comparison to adding structural typing to an existing class-based nominal system, adding nominal types to a structural type system requires only a very small change to the language, and can be achieved without any changes at all in a language with a relatively extensible type system. This is also demonstrated by the relatively simple additions of the formal model in Section 5, and by comparisons to other formalisms of branding systems.

Most nominal object-oriented type systems use a hierarchy of classes to define their types and name those types after the classes. Most branding systems use the same technique, whether or not they extend existing class-based languages. In such a language, brand types represents objects which have been created by their associated class, and conceptually include the interface of that class as well. In contrast, our brand objects have no associated class, and two objects branded with the same brand may have entirely distinct interfaces – we rely entirely on the existing structural type system to provide interface information. A brand type represents exactly those objects which have been branded by the underlying brand object, and no more.

We use three existing features of Grace in our implementation. Object annotations, where a newly constructed object is annotated with some other object, are used to explicitly brand objects (object annotations are a feature of Grace not yet discussed in the literature). A pattern [27] – which provides runtime pattern-matching facilities – acts as a brand's type, to test the presence of that brand on a given object. The dialect system [26] then allows the creation of a pluggable static type system which reasons about the patterns of brands bound to statically observable names as nominal types, and treats branded objects as inhabitants of these types.

2.1 Creating, Applying, and Using Brands

Consider a class hierarchy representing shapes, defining the concrete classes square and circle. In a strictly nominal system, an abstract class shape would form the root of this hierarchy, and create a common supertype for all concrete shape objects. In Grace, we might first define a Shape type:[1]

```
let Shape = type {
    at → Point
    area → Number
}
```

The Shape type describes the expected structure of a shape object. We could then define a class hierarchy which implements this interface, annotating the shape class as abstract with the keyword **is**:

```
class shape.at(location : Point) → Shape is abstract {
    method at → Point { location }
}
```

[1] Grace names returning types conventionally start in uppercase (Shape) while names returning objects (which may be fields, methods, or classes) start in lowercase (shape).

With concrete classes:

```
class square.at(location : Point) withLength(length : Number) → Shape {
    inherits shape.at(location)
    method area → Number { ... }
}

class circle.at(location : Point) withRadius(radius : Number) → Shape {
    inherits shape.at(location) ...
    method area → Number { ... }
}
```

Note that the classes are tagged with return types, as Grace classes are distinct from types. Moreover, all of these classes have the *same* return type, because their instances all have the same interface.

We could explicitly declare types for the objects created by the square and circle classes, by listing the signatures of the public methods in each class:

```
let Square = type { at → Point; area → Number }
let Circle = type { at → Point; area → Number }
```

These new types are identical to the Shape type defined above, and represent exactly the same set of objects. Structural types cannot distinguish between different objects with the same interface, either during static checking or at runtime.

In this paper, we introduce *brand objects* that can be used to make finer distinctions between objects – distinctions that correspond to standard nominal types. Brand objects are created by the brand method, which returns a new unique brand object. For example:

```
let aSquare = brand
```

will create a new brand object named aSquare. We can use this brand object to mark objects (e.g. those created by the square class) by annotating the class declaration with the brand:

```
class square.at(location : Point)
    withLength(length : Number) → Shape is aSquare { ... }
```

Brand objects have a Type method that returns a Grace pattern object that reifies the type of the brand. This lets us define distinct Square and Circle types by combining the structural Shape type with the types of the respective brands via Grace's type intersection operator (&).

```
let Square = Shape & aSquare.Type

let aCircle = brand
let Circle = Shape & aCircle.Type
```

Brands combined with structural interfaces produce Grace types that behave like nominal types. Square and Circle now define different types, rather than aliases of the same structural type.

This lets us go one step further: we can now declare that the square class returns an object of the Square type:

```
class square.at(location : Point)
    withLength(length : Number) → Square is aSquare { ... }
```

The instance's structural type is the same as before, but it carries the added information that it is branded as aSquare, making it an instance of the Square branded type as well. The brand aSquare is not a type, annotating the class with the brand is different from providing a return type, hence the appearance of both the aSquare brand and the Square type.

The combination of branded types with structural types follows the same type rules as other types, including subtyping. A branded object (with the appropriate brand) must be supplied where a branded object is expected:

```
def mySquare : Square = square.at(10 @ 50) withLength(20)
```

A branded object may be used anywhere an unbranded object with the same structure is expected:

```
def myShape : Shape = mySquare
```

But critically, an unbranded object cannot be supplied where a branded object is expected:

```
// Error: not an instance of Square
def myCircle : Square = circle.at(10 @ 50) withRadius(20)
```

If these declarations were repeated in another module (or even in the same module) then the repetition will create a different unique identifier and so represent a distinguishable, that is, different, brand and associated pattern object, regardless of the name the brand is bound to. The nominal aspect of the brand is its underlying object identity.

Brand objects, like types, can be reasoned about statically. For clarity on what we are treating statically in this paper we write "**let**" for all statically-known declarations, as a syntactic extension to the language. We discuss this change, which is not specific to branding, further in Section 3.

2.2 Extending Brands

Inheriting from an object that is branded causes the resulting object to have the same brand: this behavior is necessary for inheritance to preserve subtyping (required by the Grace specification [8]). Inheritance is the easiest mechanism for extending an existing brand, and provides a correspondence between class and (nominal) type, as in most nominally typed languages.

If we were to return to the shape class, and create a brand for it:

```
let aShape = brand
let Shape = aShape.Type & type { ... }
class shape.at(location : Point) → Shape is abstract, aShape { ... }
```

Now the whole shape hierarchy is branded, and the Shape type will only match objects created by the shape class, including those which inherit from it. The Square and Circle types remain subtypes of Shape and the square and circle classes inherit the aShape brand, just as if the classes were in a standard nominal typing hierarchy.

Brands need not conform to single-inheritance class hierarchies. Because brands are not inherently associated with an interface, and access to the brand object is all that is required to build an object which satisfies the brand type, any object can take advantage of multiple-subtyping without a multiple-inheritance mechanism by simply being branded with multiple brands. This is conceptually similar to a class implementing multiple interfaces in Java or C♯, providing a typing relationship without method reuse.

Brand objects also support the + operator, which creates a 'sub-brand' from two existing brands. Using this new brand is exactly the same as using the two parts together: branding an object with it causes the object to be branded with both the parts, and the brand's Type is the intersection of the patterns of both of the parts. Combining a brand with a new, anonymous brand, creates a unique sub-brand of the extended one. This behavior is included in the brand interface as the extend method.

2.3 Brands vs. Branded Types

The distinction between a brand (like aSquare) and a branded type (aSquare.Type, or Square) is crucial. Branded objects can only be created with access to the underlying brand. An untrusted object can safely be given access to the branded type, as this does not allow that object to fraudulently brand other objects. This can be achieved by exposing the branded type to other code as a public constant and keeping the brand object locally as a confidential field not accessible from the outside.

In Grace, a declared brand, like any other named declaration, is a method in an object; the name of a type (branded or not) is simply a request to the object declaring the type, and so Grace's existing visibility mechanism suffices to protect brands. In order to ensure no other class can be branded aSquare while allowing public access to the Square type, we would modify the brand declaration to be hidden:

```
let aSquare is confidential = brand
```

These declarations are public by default, so the type remains exported.

Access only flows in one direction: the brand object cannot be retrieved from the type, but access in the other direction is not limited, as the pattern object is available through the brand object with the Type method. The pattern object does not provide any privileged behavior, so it makes sense to provide uni-directional access between the objects rather than returning a pair from the brand constructor.

The three branding utilities – the brand method, the use of brands as annotations on object literals and classes, and the unique types they introduce – are the only additions Grace's structural type system requires in order to support nominal types. Moreover, using Grace's patterns and dialects, they are all achieved using existing functionality, with no brand-specific modifications to the language's syntax or semantics. In the next section, we discuss how this is achieved.

3 Implementation

We have implemented brands in Grace on top of Hopper, an existing prototype interpreter for the language. The core implementation is a single Grace module, extending the existing structural type checker described in [26]. We have also modified the language implementation to change **type** declarations to **let**, permitting any statically resolvable value to appear in the declaration.

All of the functionality specific to brands is implemented using existing features of Grace, provided in a *dialect* [26] with the necessary definitions. Every Grace module is written in a dialect that defines the methods that are in the local scope throughout that module. A dialect may also provide a check method, which is passed the AST of any module which uses it and may subsequently raise errors about the implementation of the module. The check method allows for a form of pluggable typing [13], which individual modules may opt in to. Modules written in the brand dialect will be checked, and the dynamic behavior of the

brands will work as expected in other modules, but the static checking is restricted to just those modules using the branding dialect.

Annotations are an existing feature of Grace, allowing objects to be attached to – and potentially transform – various language constructs, including singleton object constructors and classes. Hopper allows arbitrary expressions to appear in an annotation list, requiring only that the resulting object have an appropriate method for handling the construct that it annotates: for example, object annotations must have an annotateObject method. The method is implemented on brands so that it attaches the brand to the object's metadata, which is a weak set of objects associated with another object at runtime available through reflection. The metadata on a construct can be retrieved through a mirror object using the existing reflection interface. This means an object can be tested for a brand with:

```
mirrors.reflect(obj).metadata.has(aThing)
```

Grace's Pattern type, which provides an interface for testing objects against type-like objects as runtime [27], is used to build brand pattern objects. By inheriting from a standard abstract class that defines the basic implementation of patterns, the objects need only provide a concrete match method, which uses the reflection system mentioned above to inspect the metadata of the given object and discover whether the relevant brand is in the metadata set.

Internally, most of the brand object functionality is implemented in the preBrand class. This class is used to build the 'pre-brand' object aBrand. This pre-brand is local to the dialect, but the dialect makes the public type Brand for use outside of the dialect, which is aBrand's pattern object combined with the interface of brands.

```
let Brand = aBrand.Type & ObjectAnnotation & type {
    Type → Pattern
    extend → Brand
    +(other : Brand) → Brand
}
```

aBrand has the same implementation as other brands, but an object cannot appear in its own annotation list and so aBrand does not satisfy the Brand type. All other brands inherit from the same class that created aBrand, with the sole addition of being branded by aBrand, causing them to satisfy the Brand type.

```
method brand → Brand {
    object is aBrand { inherits preBrand.new }
}
```

These two definitions are included with the rest of the standard dialect definitions to provide a sensible set of default methods to any module which uses the dialect.

The dialect extends the existing structural type checker by including extra understanding of the Brand type and the result of requesting brand in its check method. Each creation of a brand object is considered distinct by the system, and this identity is tracked within the scope of its creation, as well as when it is exported by a **let** declaration. The existing structural rules are still enforced, including when structural types are paired with brand types. We formalize this combination of static typing in Section 5.

The replacement of **type** declarations with **let** is necessary to allow *any* dialect which is introducing a new type construct (rather than refining an existing one) to know what it should be reasoning about statically. This change is not specific to brands. The existing type declarations require that the value being declared be a statically-determinable structural type:

neither brands nor their pattern objects satisfy this definition, and neither will any other new value which a particular dialect treats as a static type. The definition of 'statically-known' is now determined on a module-by-module basis by the dialect a module is implemented in. The default and structural type checking dialects use the existing definition of static structural types, whereas the brand dialect extends this definition to include the new static brand values.

Matching against a brand's object identity provides no dynamic information about a brand's name, so that if a dynamic type error occurs involving a brand it cannot report the name of the brand that failed to match. Reporting type names is a standard problem in structural type systems [32] as types do not naturally have names. The use of static declarations in Grace also addresses this problem, by dynamically attaching name information to values – both brands and brand types, in our case – defined by a **let** declaration, which can be leveraged to generate better error messages. This behavior was already implemented for structural type declarations.

4 Case studies

Even in a structurally-typed language, some aspects will require more nominal semantics. This section presents applications of branding, mostly within the existing language implementation, replacing ad-hoc implementations with the branding mechanism.

4.1 Abstract Syntax Tree

An AST may contain many nodes with the same structure, but which must nevertheless be distinguished. This is a particularly important problem for Grace, as dialect check methods operate over the AST of the modules that they check. Nodes for a variable declaration and a constant definition will have a name, a value, and a type, but it is important that neither be mistaken for the other when they must be considered distinct. While the subtyping structure of an AST node is likely to be 'flat', brands allow overlaying distinguishing features on a range of otherwise-identical types.

We draw out two cases in particular from the AST of Grace source code, reflecting issues we have had ourselves in implementing the language. The **var** and **def** (variable and constant declaration) nodes mentioned in the previous paragraph have the same fundamental shape, while a **class** node has a superset of the methods of an **object** node, but is not a subtype of it. Before brands, AST nodes were 'stringly-typed', using a kind string field with the name of the node type, but this was an ad-hoc solution that sat outside of the type system. We can combine brands and types to avoid both of these issues: each kind of node now has both a structural interface and one or more nominal brands. Once we have created the relevant brands, the types can be constructed as:

```
// The common interface of both var and def nodes
let DeclNode = Node & type {
    name → String
    value → Expression
    pattern → Expression
}

let VarNode = aVarNode.Type & DeclNode
let DefNode = aDefNode.Type & DeclNode
```

The DeclNode type is purely structural, and before brands was the *only* type that applied to each of our nodes. VarNode, however, combines the structural type with the pattern of the aVarNode brand: to belong to the VarNode type, an object must have both the structural type and be branded as aVarNode.

```
class varNode.new(...) → VarNode is aVarNode { ... }

match(varNode.new(...))
    case { d : DefNode → print "A def!" }
    case { v : VarNode → print "A var!" }
```

Prior to brands, just as in our shapes example from earlier, the VarNode statement would ‘fall into’ the DefNode branch [11], because it is the first to appear and the structural type would match, and similarly a **def** node could be passed to a method expecting a **var** node without error. With brands, the DefNode branch does not match and the correct branch is given an opportunity to match, while both static and dynamic type checks will behave as desired.

4.2 Dialects

Dialects can be defined by expressing the checking as a series of rule blocks [26]. Rule blocks specify which nodes they apply to by typing their input, but this presents a problem to the type checker: if the input is stringly-typed, the type checker cannot determine what the type means and so cannot check the body of the rule. Misuses such as the spelling error below will not be caught until runtime, despite the rule being annotated with what appear to be types.

```
rule { vn : VarNode →
    if (vn.vallue.isEmpty) then {
        CheckerFailure.raise("All vars must be assigned to") forNode(vn)
    }
}
```

The extended reasoning of the branding allows the type checker to understand the combination of structural and nominal type.

A similar issue can occur when one type is a structural supertype of another. This situation arises in the case of class and object nodes, and the same resolution can be applied:

```
let ObjectNode = anObjectNode.Type & type {
    body → List<Node>
}

let ClassNode = aClassNode.Type & type {
    body → List<Node>
    name → String
}

class classNode.new(...) → ClassNode is aClassNode { ... }
```

Objects created by classNode will not be considered to belong to the ObjectNode type, notwithstanding that they possess all of the methods of object nodes. Before brands these nodes were distinguished by string fields found in all nodes, outside of the type system. Using

fields in this way is clearly suboptimal, particularly as it sits outside the protection of the type system.

4.3 Exceptions

Representations of runtime errors encode a degree of hierarchy, and must be both created and caught within this hierarchy. For example, a FileNotFoundError may be a specialization of IOError, which is itself a RuntimeError. An exception handler must be able to declare it wishes to trap all IOErrors, including specializations. In a nominal language such as Java this behavior maps naturally onto nominal class inheritance, with a handler for one exception type implicitly trapping all its subtypes by subsumption. In a structurally-typed language this relationship does not exist innately and must be created.

Grace's explicit exception hierarchy leverages the pattern-matching system for handlers. An *exception kind* is an object representing a kind of exception, and includes two methods. The refine method creates a new exception kind as a child of the receiver. The raise method creates an exception object, which is propagated up the stack until a handler is reached. All exception kind objects are patterns, matching any exception packet derived from itself or its refined descendants.

```
def FileNotFoundError = IOError.refine("File not found")
try {
    if (!exists(path)) then {
        FileNotFoundError.raise("{path} does not exist")
    }
} catch { e : IOError →
    print "An IO error occurred: {e}"
}
```

The catch block above will trap the exception raised in the try block because the exception kind FileNotFound was refined from IOError.

This system is reminiscent of brands and can be placed on firmer footing through their use. An ExceptionKind's match method delegates to the Type object of a brand, and its raise method creates an appropriately-branded exception packet. Omitting implementation details, the structure of the exception kind hierarchy can look like the following:

```
class exceptionKind.name(name : String) brand(aKind : Brand) → ExceptionKind {
    ...
    method refine(name : String) → ExceptionKind {
        exceptionKind.name(name) brand(aKind.extend)
    }
    method raise(message : String) → None {
        internal.raise(object is aKind { inherits exception })
    }
    method match(obj : Object) → MatchResult {
        aKind.Type.match(obj)
    }
}

let Exception = exceptionKind.name("Exception") brand(brand)
```

In this way brands provide a well-founded structure for an existing sui generis construct of the language. An exceptional behavior has been replaced with a consistent general-purpose approach that can be applied in user code elsewhere.

4.4 Singleton types

A singleton type is a type with only a single element, which may or may not be trivial. Singleton types are one way of adding nominal types into a structural language,[2] but we find it more advantageous to go in the other direction: to use brands as the means to add singleton types to a language without them. If a sentinel value unit is defined as an empty object, then its structural type is **type** {}, which is inhabited by every object. If Unit is to be a proper unit type, with unit as its only inhabitant, then we can define:

```
let theUnit is confidential = brand
let Unit = theUnit.Type

def unit is public = object is theUnit {}
```

As theUnit is not publicly available, other modules cannot brand other objects with it, and so unit will always be the only inhabitant proper of Unit.

Similarly, the empty type can also be constructed by taking the pattern of an anonymous brand, ensuring that no object can ever be branded by it and, by extension, ever be an instance of the resulting type.

```
let None = brand.Type
```

A brand need not be bound to a name to take its Type.

5 Formal Model

In this section, we model branded types by extending Tinygrace, an existing formal model of a subset of the Grace language [29]. A Tinygrace program is a set of type declarations and an expression to be evaluated. Unlike the gradual type system of the full Grace language, type information is always required: Tinygrace types are mandatory, and there is no Unknown (dynamic) type. Our extension makes two simple additions: branding objects with the **is** annotation marker, and creating brands with the brand declaration, as in the Grace implementation.

Brand declarations have no associated name or structural type information, but their associated types may be combined with structural information using the combinator &, and must be bound to a name or combined with other brands in order to be useful. In the following figures, changes to the existing model not related to structural typing are highlighted.

5.1 Syntax

The abstract syntax for the model is defined in Figure 1. The metavariable T ranges over static expression declarations; M over methods; O over object literals; C over case

[2] We discuss this approach in Section 6.

Syntax

$P ::= \overline{T}\ e$ *(Program)*

$M ::=$ **method** $S\ \{\, e\, \}$ *(Method)*

$O ::=$ **object** **is** $\overline{B}$ $\{\, \overline{M}\, \}$ *(Object constructor)*

$C ::=$ **match**(e) $\overline{\textbf{case}\,\{\, x : \tau \to e\, \}}$ *(Match-Case branch)*

$\tau\ ::=$ **type** $\{\, \overline{S}\, \} \mid \mu X.\tau \mid X \mid (\tau \mid \tau) \mid (\tau\ \&\ \tau) \mid B.\mathsf{Type}$ *(Type)*

$S\ ::= m(\overline{x : \tau}) \to \tau$ *(Method signature)*

$e\ ::= x \mid e.m(\overline{e}) \mid O \mid C$ *(Expression)*

$B ::= \mathsf{brand} \mid B + B \mid X \mid \beta$ *(Brand expression)*

$E ::= \tau \mid B$ *(Static expression)*

$T ::=$ **let** $X = E$ *(Static declaration)*

Contexts

$\Sigma ::= \cdot \mid \Sigma,\ X <: Y$ *(Subtyping context)*

$\Gamma\ ::= \cdot \mid \Gamma,\ x : \tau$ *(Typing context)*

Auxiliary Definitions

$\mathsf{or}(\tau) = \tau$
$\mathsf{or}(\tau,\ \overline{\tau}) = \tau \mid \mathsf{or}(\overline{\tau})$

$\mathsf{and}(\tau) = \tau$
$\mathsf{and}(\tau,\ \overline{\tau}) = \tau\ \&\ \mathsf{and}(\overline{\tau})$

Figure 1 Grammar for Tinygrace with branding extension.

expressions; x and y over variable names; X and Y over static expression alias names; τ over type expressions; S over method signatures, m over method names, and e over expressions.

We write $\overline{e}$ to indicate a possibly empty sequence of comma-separated expressions $e_1, \ldots, e_n$, as well as for method signature parameters $\overline{x : \tau}$ and type names $\overline{X}$, hiding the parentheses and **is** keywords when there are no values for them to delimit. We also write $\overline{S}$, $\overline{T}$, and $\overline{M}$ to indicate a possibly empty set of declarations $S_1 \ldots S_n$, $T_1 \ldots T_n$, and $M_1 \ldots M_n$ respectively, and $\overline{\textbf{case}\,\{\, x : \tau \to e\, \}}$ (or just $\overline{C}$) to indicate a non-empty sequence of case branches. As a Grace module is just an object, but the model does not allow static declarations in object bodies, a program is any pair of the form $\overline{T}\ e$. We follow Tinygrace in using Barendregt's variable convention [4, 44] that bound and free variables are distinct.

The set of type declarations $\overline{T}$ allows type aliasing as well as a mechanism for expressing recursive types without explicit folding. The declarations are resolved to explicitly recursive μ-types and substituted throughout the program to remove all aliases at runtime. μ-types are not a part of the concrete syntax, and can only arise from the normalization of type declarations.

$\boxed{\overline{T}\ e\ \triangleright\ e}$

$$\text{(N-Prog)}\quad \frac{\overline{T} \vdash \overline{T} \triangleright \overline{E'}}{\overline{T}\ e \triangleright [\overline{X \mapsto E'}]e} \quad \overline{T} = \overline{\textbf{let}\ X = E}\text{, with } \overline{X} \text{ distinct}$$

$\boxed{\overline{T} \vdash \textbf{let}\ X = E\ \triangleright\ E'}$

$$\text{(N-Type-Decl)}\quad \frac{\overline{T} \vdash \tau \checkmark}{\overline{T} \vdash \textbf{let}\ X = \tau \triangleright \mu X.\tau} \quad \mu X.\tau \text{ contractive}$$

$$\text{(N-Brand-Decl)}\quad \frac{\overline{T} \vdash B \triangleright B'}{\overline{T} \vdash \textbf{let}\ X = B \triangleright B'}$$

$\boxed{\overline{T} \vdash B\ \triangleright\ B'}$

$$\text{(N-Brand)}\quad \frac{}{\overline{T} \vdash \mathsf{brand} \triangleright \beta} \quad \beta \text{ fresh}$$

$$\text{(N-Name)}\quad \frac{}{\overline{T} \vdash X \triangleright X} \quad \textbf{let}\ X = B \in \overline{T}$$

$$\text{(N-Plus)}\quad \frac{\overline{T} \vdash B_1 \triangleright B_1' \qquad \overline{T} \vdash B_2 \triangleright B_2'}{\overline{T} \vdash B_1 + B_2 \triangleright B_1' + B_2'}$$

Figure 2 Declaration normalization.

The major syntactic addition is the brand expression, with the metavariable B. The concrete syntax includes the brand constructor, sums of brands, and references to static brand declarations. Individual brands are resolved to a unique value in the set of names β. Like μ-types, β names are not part of the concrete syntax, arising only from the resolution of brand constructors. Ultimately, the names that brands are bound to become irrelevant, and they are identified solely by the uniqueness of their β name.

The remaining additions involve usage of brands: annotated objects and references to a brand's Type. The new metavariable E ranges over both brand expressions and types, allowing static declarations to refer to either.

5.2 Well-Formedness and Normalization of Declarations

The normalization judgments for programs, static declarations, and methods are given in Figure 2. A program $\overline{T}e$ is normalized into a single expression e' by $\overline{T}e \triangleright e'$ before it is analyzed or reduced. This normalization procedure corresponds to the well-formedness judgments of Tinygrace, making explicit how type declarations are transformed into anonymous recursive types inside the program expression.

Brand declarations B are normalized by $\overline{T} \vdash B \triangleright B'$, removing all occurrences of the brand constructor and replacing them with unique β identifiers to produce B'. Brand normalization also ensures well-formedness, as names in brand declarations must refer to an

$\boxed{\overline{T} \vdash \tau \checkmark}$

$$\text{(W-STRUCT)}\ \frac{\overline{T} \vdash \overline{\tau_p \checkmark} \qquad \overline{T} \vdash \tau_r \checkmark}{\overline{T} \vdash \textbf{type}\,\{\, \overline{m(\overline{x : \tau_p}) \to \tau_r}\,\} \checkmark}\ \overline{m}\ \text{distinct} \qquad \text{(W-NAME)}\ \frac{}{\overline{T} \vdash X \checkmark}\ \textbf{let}\ X = \tau \in \overline{T}$$

$$\text{(W-AND)}\ \frac{\overline{T} \vdash \tau_1 \checkmark \qquad \overline{T} \vdash \tau_2 \checkmark}{\overline{T} \vdash \tau_1 \,\&\, \tau_2 \checkmark} \qquad \text{(W-OR)}\ \frac{\overline{T} \vdash \tau_1 \checkmark \qquad \overline{T} \vdash \tau_2 \checkmark}{\overline{T} \vdash \tau_1 \mid \tau_2 \checkmark}$$

$$\text{(W-REC)}\ \frac{\overline{T}, \textbf{let}\ X = \tau \vdash \tau \checkmark}{\overline{T} \vdash \mu X.\tau \checkmark} \qquad \text{(W-BRAND)}\ \frac{\overline{T} \vdash B \triangleright B'}{\overline{T} \vdash B.\mathsf{Type} \checkmark}$$

$\boxed{\Gamma \vdash M \checkmark}$

$$\text{(W-METH)}\ \frac{\Gamma, \overline{x : \tau_p} \vdash e_r : \tau_r}{\Gamma \vdash \textbf{method}\ m(\overline{x : \tau_p}) \to \tau_r\ \{\, e_r \,\} \checkmark}$$

Figure 3 Type and method well-formedness.

existing brand declarations in order to normalize. Note that normalization *only* transforms named declarations of brands, and other appearances of brand that appear in types or in the program expression are not resolved to some β. These 'dangling' brands can persist throughout the type-checking and execution of a program, but are considered distinct by subtyping and so cannot have any adverse effect on any of the remaining judgments.

Each type declaration must not resolve to itself (**let** $X = X$), and its interpretation as a tree must be *contractive*.

▶ **Definition 1.** A type tree is contractive if it corresponds to a finite series of μ-types or applications of & or $|$, so that all paths traversing any of those three operations terminates in a type literal.

For a type $\mu X.\tau$, this means that X may not appear in τ – either directly or through a reference to some other, mutually recursive declaration in $\overline{T}$ – except inside the body of a structural type literal. In the concrete syntax, this extends the set of invalid types to include declarations such as **let** $X = X \,\&\, Y$, or **let** $X = Y$; **let** $Y = X$.

Programs normalize to some expression e' when their set of declarations (both types and brands) normalize, and e' is the result of substituting the normalized declarations into the expression, with e' well-typed.

Type and method well-formedness are defined in Figure 3. Type well-formedness, $\overline{T} \vdash \tau \checkmark$, ensures that names inside τ only refer to existing type declarations. References to a brand's Type normalizes the brand, but discards the result, as only the well-formedness of the brand is required for the type to be well-formed. Dangling brands in Type references are retained, causing the whole type to be empty.

The well-formed judgment for methods $\Gamma \vdash M \checkmark$ just defers to the type judgment of the body of M in the scope of the method parameters.

5.3 Subtyping

The rules for type and signature subtyping are given in Figure 4, and are mostly standard. The subtyping relation is written $\Sigma \vdash \tau_1 <: \tau_2$, meaning a type τ_1 is a subtype of type τ_2 in the context of the assumption set Σ. The primary subtyping rule is Rule S-Struct, which states that two structural types $\textbf{type}\,\{\,\overline{S_1}\,\}$ and $\textbf{type}\,\{\,\overline{S_2}\,\}$ are in the subtyping relationship if, for a signature S_2, there is a signature S_1 with matching parameter types and return type in contravariant and covariant relationships respectively, and the subtraction of the matching signature from each type are also subtypes in the same direction. This forms an algorithmic approach to structural subtyping, removing a signature one at a time before terminating at Rule S-Top.

The assumption set models the otherwise coinductive nature of the recursive subtyping in a well-founded inductive setting. When comparing two recursive types with Rule S-Unfold the types may be unfolded but a subtyping relationship between the names bound by μ is added to the assumption set. If the same relationship is compared again, it succeeds through Rule S-Assum, modeling the infinite repetition permitted by coinduction. The assumption set need only contain the names of the types, as the well-formedness rules ensure that bound type names are unique to the type: if $\mu X.\tau$ appears in the program, any appearance of $\mu X.\tau'$ is guaranteed to have $\tau' = \tau$. Because the rules are interpreted inductively, the transitivity expressed in Rule S-Trans does not cause the judgment to degenerate.

The partial unfolding Rules S-Unfold-Left and S-Unfold-Right do not add to the assumption set, but the contractivity rules from the well-formedness rules ensure that the unfolded variable does not appear except inside of a type literal. This means that Rule S-Struct must apply in between, advancing the judgment.

The union and intersection types follow standard subtyping rules. The rules for subtyping of & are not complete with respect to the full language, as $\textbf{type}\,\{\,\overline{S_1}\,\}\ \&\ \textbf{type}\,\{\,\overline{S_2}\,\}$ is not equivalent to the union of $\overline{S_1}$ and $\overline{S_2}$ (an operation that requires combining signatures with the same name). This incompleteness does not affect the subset we are modeling.

The branding extension adds three rules to the subtyping judgment. Types of equivalent β names are subtypes, providing reflexivity for brands, and the patterns of sums of brands just defer to the intersection of the patterns of the summed brands. These rules only affect the relevant proofs by adding these extra cases into inductive reasoning.

The instance judgment $O \in \tau$, defined in Figure 5, means that a concrete object O is in the type τ. The judgment is defined directly in terms of subtyping on the concrete type of O, which the branded addition extends to include all of the brand patterns with the & combinator (the and auxiliary function is defined in Figure 1). We can interpret a type τ as a set $[\![\tau]\!]$ containing all of the concrete objects that are instances of that type, $\{O \mid O \in \tau\}$. We prove soundness of subtyping with respect to these set semantics.

First we require an inversion lemma on the instance judgment.

▶ **Lemma 2.** *If* $\textbf{object}\ \textbf{is}\ \overline{B}\ \{\overline{\textbf{method}\ S\ \{\,e\,\}}\} \in \tau$, *then* $\cdot \vdash \mathsf{and}(\textbf{type}\,\{\,\overline{S}\,\}, \overline{B.\mathsf{Type}}) <: \tau$

Proof. Trivial inversion of $O \in \tau$ by Rule S-In. ◀

We can now show soundness of subtyping, with respect to the set semantics of the types.

▶ **Theorem 3.** *If* $\cdot \vdash \tau_1 <: \tau_2$, *then* $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$.

$\boxed{\Sigma \vdash \tau_1 <: \tau_2}$

(S-Struct)
$$\frac{\Sigma \vdash \overline{\tau_{p2} <: \tau_{p1}} \qquad \Sigma \vdash \tau_{r1} <: \tau_{r2} \qquad \Sigma \vdash \textsf{type}\,\{\,\overline{S_1}\,\} <: \textsf{type}\,\{\,\overline{S_2}\,\}}{\Sigma \vdash \textsf{type}\,\{\,m(\overline{x_1 : \tau_{p1}}) \to \tau_{r1}\ \overline{S_1}\,\} <: \textsf{type}\,\{\,m(\overline{x_2 : \tau_{p2}}) \to \tau_{r2}\ \overline{S_2}\,\}}$$

(S-Top)
$$\frac{}{\Sigma \vdash \tau <: \textsf{type}\,\{\}}$$

(S-Assum)
$$\frac{}{\Sigma \vdash \mu X.\tau_1 <: \mu Y.\tau_2}\ X <: Y \in \Sigma$$

(S-Unfold)
$$\frac{\Sigma, X <: Y \vdash [X \mapsto \mu X.\tau_1]\tau_1 <: [Y \mapsto \mu Y.\tau_2]\tau_2}{\Sigma \vdash \mu X.\tau_1 <: \mu Y.\tau_2}\ X <: Y \notin \Sigma$$

(S-Unfold-Left)
$$\frac{\Sigma \vdash [X \mapsto \mu X.\tau]\tau <: \textsf{type}\,\{\,\overline{S}\,\}}{\Sigma \vdash \mu X.\tau <: \textsf{type}\,\{\,\overline{S}\,\}}$$

(S-Unfold-Right)
$$\frac{\Sigma \vdash \textsf{type}\,\{\,\overline{S}\,\} <: [X \mapsto \mu X.\tau]\tau}{\Sigma \vdash \textsf{type}\,\{\,\overline{S}\,\} <: \mu X.\tau}$$

(S-Or-Left)
$$\frac{\Sigma \vdash \tau <: \tau_1}{\Sigma \vdash \tau <: \tau_1 \mid \tau_2}$$

(S-Or-Right)
$$\frac{\Sigma \vdash \tau <: \tau_2}{\Sigma \vdash \tau <: \tau_1 \mid \tau_2}$$

(S-Or)
$$\frac{\Sigma \vdash \tau_1 <: \tau \qquad \Sigma \vdash \tau_2 <: \tau}{\Sigma \vdash \tau_1 \mid \tau_2 <: \tau}$$

(S-And)
$$\frac{\Sigma \vdash \tau <: \tau_1 \qquad \Sigma \vdash \tau <: \tau_2}{\Sigma \vdash \tau <: \tau_1\ \&\ \tau_2}$$

(S-And-Left)
$$\frac{\Sigma \vdash \tau_1 <: \tau}{\Sigma \vdash \tau_1\ \&\ \tau_2 <: \tau}$$

(S-And-Right)
$$\frac{\Sigma \vdash \tau_2 <: \tau}{\Sigma \vdash \tau_1\ \&\ \tau_2 <: \tau}$$

(S-Trans)
$$\frac{\Sigma \vdash \tau_1 <: \tau_2 \qquad \Sigma \vdash \tau_2 <: \tau_3}{\Sigma \vdash \tau_1 <: \tau_3}$$

(S-Brand-Refl)
$$\frac{}{\Sigma \vdash \beta.\textsf{Type} <: \beta.\textsf{Type}}$$

(S-Brand-Left)
$$\frac{\Sigma \vdash B_1.\textsf{Type}\ \&\ B_2.\textsf{Type} <: \tau}{\Sigma \vdash (B_1 + B_2).\textsf{Type} <: \tau}$$

(S-Brand-Right)
$$\frac{\Sigma \vdash \tau <: B_1.\textsf{Type}\ \&\ B_2.\textsf{Type}}{\Sigma \vdash \tau <: (B_1 + B_2).\textsf{Type}}$$

Figure 4 Subtyping judgment.

Proof. Take any O such that $O \in [\![\tau_1]\!]$. For the exact type τ_o of O, $\cdot \vdash \tau_o <: \tau_1$ by Lemma 2. As $\cdot \vdash \tau_1 <: \tau_2$, $\cdot \vdash \tau_o <: \tau_2$ by the transitivity of subtyping, so $O \in \tau_2$ by Rule S-In. ◀

This property is not particularly interesting, as the instance rule is defined directly in terms of subtyping and so these outcomes are relatively obvious. We are more interested in the property of *method inclusion*, which ensures that when an expression is typed through the subtyping property, the resulting concrete object will have the necessary methods indicated by the type of the expression – this property is necessary for a structural subtyping system to be sound, and a proof for progress of well-typed terms under reduction relies on it. As not all types are structural, we consider any set of signatures in a structural supertype (equivalent to using a subsumption rule, which will be defined later).

$\boxed{O \in \tau}$

$$\text{(S-In)}\ \frac{\cdot \vdash \mathsf{and}(\textbf{type}\,\{\,\overline{S}\,\}, \overline{B.\mathsf{Type}}) <: \tau}{\textbf{object}\ \ \textbf{is}\ \overline{B}\ \ \{\,\overline{\textbf{method}\ S\ \{\,e\,\}}\,\} \in \tau}$$

Figure 5 Instance judgment.

$\boxed{e \longmapsto e'}$

$$\text{(R-Recv)}\ \frac{e \longmapsto e'}{e_s.m(\overline{e_p}) \longmapsto e'.m(\overline{e_p})}$$

$$\text{(R-Prm)}\ \frac{e \longmapsto e'}{O_s.m(\overline{O_p}, e, \overline{e_p}) \longmapsto O_s.m(\overline{O_p}, e', \overline{e_p})}$$

$$\text{(R-Req)}\ \frac{\textbf{method}\ m(\overline{x : \tau_p}) \to \tau_r\ \{\,e_r\,\} \in \overline{M}}{\textbf{object}\ \ \textbf{is}\ \overline{B}\ \ \{\,\overline{M}\,\}.m(\overline{O_p}) \longmapsto [\mathsf{self} \mapsto \textbf{object}\ \ \textbf{is}\ \overline{B}\ \ \{\,\overline{M}\,\}, \overline{x \mapsto O_p}]e_r}\ \ |\overline{x}| = |\overline{O_p}|$$

$$\text{(R-Match)}\ \frac{e \longmapsto e'}{\textbf{match}(e)\ \overline{\textbf{case}\,\{\,x : \tau \to e_c\,\}} \longmapsto \textbf{match}(e')\ \overline{\textbf{case}\,\{\,x : \tau \to e_c\,\}}}$$

$$\text{(R-Case)}\ \frac{}{\textbf{match}(O)\ \textbf{case}\,\{\,x : \tau \to e_c\,\} \cdots \longmapsto [x \mapsto O]e_c}\ \ O \in \tau$$

$$\text{(R-Miss)}\ \frac{}{\textbf{match}(O)\ \textbf{case}\,\{\,x : \tau \to e_c\,\}\ \overline{C} \longmapsto \textbf{match}(O)\ \overline{C}}\ \ O \notin \tau$$

Figure 6 Small-step semantics of reduction.

▶ **Theorem 4.** *If* $\cdot \vdash \tau <: \textbf{type}\,\{\,\overline{S_1}\,\}$, *then for any* $\textbf{object}\,\{\,\overline{\textbf{method}\ S_2\ \{\,e\,\}}\,\} \in [\![\tau]\!]$, *every* $S_1 \in \overline{S_1}$ *has a corresponding method* $S_2 \in \overline{S_2}$ *such that* $\cdot \vdash S_2 <: S_1$.

Proof. By induction over the derivation of $\cdot \vdash \tau <: \textbf{type}\,\{\,\overline{S}\,\}$. All of the rules which do not permit a structural type literal on the right of the subtyping judgment are irrelevant, and cannot become relevant in the derivation (because none of the relevant rules apply subtyping with a different type on the right side), so the derivation must ultimately terminate at Rule S-Top, by removing all of the signatures in the structural supertype after finding compatible signatures in the subtype through applications of Rule S-Struct. With case analysis on the last step, Rule S-Top is trivial, and the remaining relevant rules follow directly from the induction hypothesis. ◀

5.4 Semantics

The rules for the reduction relation $\longmapsto$ are given in Figure 6. The relation is written $e \longmapsto e'$, meaning that the expression e reduces to e' in a single reduction step. We write $\longmapsto^*$ for the

$\boxed{\Gamma \vdash e : \tau}$

$$\text{(T-VAR)}\quad \frac{}{\Gamma \vdash x : \tau}\; x : \tau \in \Gamma \qquad \text{(T-SUB)}\quad \frac{\Gamma \vdash e : \tau_1 \qquad \cdot \vdash \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}$$

$$\text{(T-OBJ)}\quad \frac{\cdot \vdash \textbf{type}\,\{\,\overline{S}\,\} \checkmark \qquad \Gamma, \mathsf{self} : \mathsf{and}(\textbf{type}\,\{\,\overline{S}\,\}, \overline{B.\mathsf{Type}}) \vdash \overline{\textbf{method}\ S\ \{\,e\,\}} \checkmark}{\Gamma \vdash \textbf{object}\ \ \textbf{is}\ \overline{B}\ \ \{\,\overline{\textbf{method}\ S\ \{\,e\,\}}\,\} : \mathsf{and}(\textbf{type}\,\{\,\overline{S}\,\}, \overline{B.\mathsf{Type}})}$$

$$\text{(T-REQ)}\quad \frac{\Gamma \vdash e_s : \textbf{type}\,\{\,\overline{S}\,\} \qquad m(\overline{x : \tau_p}) \to \tau_r \in \overline{S} \qquad \Gamma \vdash \overline{e_p : \tau_p}}{\Gamma \vdash e_s.m(\overline{e_p}) : \tau_r}\; |\overline{x}| = |\overline{e_p}|$$

$$\text{(T-CASE)}\quad \frac{\Gamma \vdash e : \mathsf{or}(\overline{\tau_p}) \qquad \overline{\Gamma, x : \tau_p \vdash e_c : \tau_c}}{\Gamma \vdash \textbf{match}(e)\ \overline{\textbf{case}\,\{\,x : \tau_p \to e_c\,\}} : \mathsf{or}(\overline{\tau_c})}$$

Figure 7 Term typing judgment.

reflexive and transitive closure of $\longmapsto$. Object constructors are the only normal form of any expression. The judgment $e \longmapsto^* O$ represents a successful execution.

The rules for typing of terms is given in Figure 7. The typing judgment for expressions has the form $\Gamma \vdash e : \tau$, meaning that in the variable environment Γ, e has the type τ. The branding extension only modifies one rule that, like the instance judgment extension, folds the & combinator over a branded object's structural type and its declared brands' Types. We prove the soundness of the system through standard progress and preservation [22, 34], beginning with progress:

▶ **Theorem 5.** *If $\cdot \vdash e : \tau$, then $e \longmapsto e'$ or e is a term of the form O.*

Proof. By induction on the derivation of $\cdot \vdash e : \tau$, with a case analysis on the last step. Rule T-VAR is irrelevant, as $\Gamma = \cdot$. Rule T-SUB and Rules T-REQ and T-CASE for congruence follow from the induction hypothesis. Rule T-REQ for computation ensures that an appropriate method with appropriate parameter cardinality exists (guaranteed through subsumption with Theorem 4), and Rule T-CASE for computation ensures that either there is more than one case or, when there is one case, the object is guaranteed to match that final case. The added branding rule in Rule T-OBJ is as trivial as the existing object rule, as a branded object is still a value and so is still in normal form. ◀

Next we require substitution preservation.

▶ **Lemma 6.** *If $\Gamma, x : \tau' \vdash e : \tau$ and $\Gamma \vdash O : \tau'$, then $\Gamma \vdash [x \mapsto O]e : \tau$.*

Proof. By induction on the derivation of $\Gamma, x : \tau' \vdash e : \tau$. The cases are straightforward examinations of substitution. Branding only affects objects, which may have brand annotations, but this is not affected by substitution. ◀

The following lemma takes the observation that, for the subtyping relationship $\cdot \vdash \tau_1 <: \mathsf{or}(\overline{\tau_2})$, at least one of the types in $\overline{\tau_2}$ must also be a supertype of τ_1, and places it in the context of preservation for Rule R-Miss, where subtracting a type from the series of unions preserves the union as a supertype of τ_1 if the subtracted type was not a supertype of τ_1.

▶ **Lemma 7.** *If* $\cdot \vdash \tau_1 <: \tau_2 \mid \tau_3$ *and* $\cdot \nvdash \tau_1 <: \tau_2$*, then* $\cdot \vdash \tau_1 <: \tau_3$*.*

Proof. By induction on the derivation of $\cdot \vdash \tau_1 <: \tau_2 \mid \tau_3$. Rule S-Or-Left cannot apply, and the remaining rules must ultimately delegate to Rule S-Or-Right to be well-founded. Branding just adds a case for Rule S-Brand-Left. ◀

Now we have the tools for a proof of preservation:

▶ **Theorem 8.** *If* $\cdot \vdash e : \tau$ *and* $e \longmapsto e'$*, then* $\cdot \vdash e' : \tau'$ *where* $\cdot \vdash \tau' <: \tau$*.*

Proof. By induction on the derivation of $e \longmapsto e'$. The congruence rules follow straightforward induction, and the computation rules are derived from the two previous lemmas, using Lemma 6 for Rules R-Req and R-Case, and Lemma 7 for Rule R-Miss. Branding has a minimal impact on both reduction and typing of terms, and so does not pose a problem here. ◀

And finally we have type soundness.

▶ **Theorem 9.** *If the expression* e *is well-typed with* $\cdot \vdash e : \tau$*, and the reduction* $e \longmapsto^* e'$ *results in* e' *a normal form, then* e' *is in the form* O *where* $\cdot \vdash O : \tau'$ *with* $\cdot \vdash \tau' <: \tau$*.*

Proof. Type soundness follows immediately from the two previous theorems. ◀

6 Discussion

In order to demonstrate the *relative* simplicity of our branding additions, we compare our formal model to formalizations of the other branding systems Unity [32], and the Tagging Language [25]. Objective measures of the complexity of type systems are difficult, but we can produce a simple comparison on the number of formal rules that are not significant to branding, alongside the rules that are. The outcome is the table in Figure 8. We omit Whiteoak [24] because it does not provide a formal model, but the Whiteoak design has almost as many additions to the syntax as Tinygrace makes overall. Both Tinygrace and the branded extension have a relatively large well-formedness overhead, but they have the same number of subtyping rules as Unity, and brands make a substantially smaller impact on typing and reduction in Tinygrace than in either Unity or even the significantly smaller Tagging Language (which lacks objects, fields, classes, and dynamic dispatch). The larger number of well-formedness rules in Tinygrace seems to stem from making less assumptions in the translation from type (and brand) declarations to recursive type (and brand) expressions.

Another option to support nominal types in any structural system by the addition of unique method names into the type, and empty implementations of these methods into the objects which are expected to fulfill this type. While the ‘phantom method’ approach works in theory, it is difficult to implement in a way that preserves the necessary encapsulation goals of nominal typing. If the methods are provided manually, the developer must provide method names that will not be used anywhere else in the program, and the encapsulation is trivially bypassed by adding the appropriate methods to other, external objects. If the methods are produced automatically, then either the branding mechanism cannot be private, or the automatic names must be indexed by something (presumably the module in which

	Tinygrace	Unity	Tagging
Syntax	7 + 4	9 + 5	5 + 5
Well-formedness	8 + 5	4 + 2	3 + 2
Subtyping	13 + 3	13 + 3	2 + 2
Term typing	5 + 1	9 + 2	6 + 4
Reduction	7 + 0	14 + 4	3 + 4
Total	40 + 13	49 + 16	19 + 17

Figure 8 Comparison of rule modifications required by branding.

the branding appears), in which case brands cannot be shared because no other module may perform the same branding. A unique singleton or empty type as the return type of a common 'branding' method in a type is another approach, but it suffers from the same problem in that the brand and type cannot be exposed separately. Neither mechanism can simultaneously service both public construction and private implementation. Our brands, in contrast, provide a fine-grained mechanism for providing access to the branding mechanism components.

Branding provides a partial solution to Boyland's gradual guarantee in gradually typed code [11], as testing an object against a brand pattern at runtime is always definitive, and is not affected by type annotations. As an extension, branding is not pervasive among objects, and so using brand patterns is only applicable to objects which have been explicitly branded. Removing reified types from the language (another proposed solution) while retaining the existing object instance rules would remove the consistency of brand patterns (as both static entities and runtime objects) alongside structural types.

Compared to standard nominal class declarations, the branding mechanism is necessarily verbose, requiring a manual separation of the brand from its type (mirroring the separation between classes and structural types in Grace). This verbosity is mostly a product of the fact that brands have been implemented without modifying the language syntax or semantics, but it also serves a purpose in demonstrating that it is not the natural mechanism for typing in Grace: structural typing is sufficient for most purposes, and it is only special cases (as seen in Section 4) where the manual separation of brand and pattern that branding should apply. It is conceivable that a more terse mechanism for direct class/type declarations could exist:

```
class Shape.new is nominal { ... }
```

Adding nominal classes directly defies Grace's design goal of maintaining a separation of type and implementation, however [6].

7 Related Work

The dichotomy between structural and nominal subtyping has been studied from the earliest applications of types to object-oriented languages [10]. Simula, the first object-oriented language, is nominally typed: a subclass must be explicitly declared as inheriting (being prefixed) by its superclass [5]. Most object-oriented languages (C++, Java, C♯, etc) followed Simula's lead, although OCaml [31] supports structural subtyping for objects, as does Go [41].

Most early theoretical analysis of type systems for object-oriented languages used structural types [20, 17, 15, 40]. Later references such as Palsberg and Schwartzbach [38], Bruce [16], and Pierce [39] discuss structural and nominal (sub)typing, but they do not address the question of how both kinds of types can best be integrated into a single, practical, language design.

Our notion of nominal brands on structural types originated in Modula-3 [35]. Record types in Modula-3 generally use structural equivalence, but can be annotated with a brand to give nominal equivalence. Modula-3 brands can also be given explicitly, e.g. for type safety between programs or across networks. Even with structural equivalence, Modula-3 record types do not support subtyping: there is no type relationship between a record type with a particular set of fields, and a second record type with a subset (or superset) of those fields – only between two record types whose field types are identical. Modula-3's object types are "essentially SIMULA classes" [18] and, like SIMULA, use nominal subtyping. Neither Cardelli et al.'s formalization of the Modula-3 type rules [18], nor Abadi's Baby Modula-3, [1], nor the *Theory of Objects* [2] model Modula-3's branded types: rather, the formal model we present here is the first we know of that shows how Modula-3–inspired brands can allow a language to support both structural and nominal subtyping.

Malayeri and Aldrich's Unity language [32, 33] is a more recent clean-sheet language design that aims to support both nominal and structural typing. Unity also uses brands to support nominal typing, but brands in Unity are essentially nominal classes – unlike Modula-3 or Grace brands, which are annotations on structural types and objects respectively. Unity's brands define the core object hierarchy in a Unity program, potentially extending a superbrand and defining fields and methods in the exactly same way that in, say, Java classes potentially extend a superclass and define the fields and methods of their instances. Unity objects are created by instantiating a brand, again just as Java (or SIMULA) objects are created by instantiating a class. Brands give Unity a nominal core, to which structural types are then added, in contrast to the approach presented here, which adds nominal brands on top of structural types. Unity draws on structural types to support external multimethods and fields defined outside objects. While Grace does not not support multimethods, similar effects can often by obtained by pattern matching, which Grace supports using both structural and nominal types. Unity was modeled formally, but not implemented.

Glew's Tagging Language [25] introduces 'tags', which are conceptually very similar to our brand objects, in the context of type dispatch. Tags can be used to implement class- and exception-casing in much the same way as brands. The underlying type system is not structural, and is populated by primitive sequence and function types instead. The language formalism goes into depth on the existence of tags at runtime, including populating the heap and runtime matching.

Gil and Maman's Whiteoak [24] in many ways takes a more pragmatic approach than Unity to combining structural and nominal types. Where Unity is a clean-sheet design, Whiteoak adds structural types to Java. Whiteoak uses the '`struct`' keyword (reserved in Java) to define structural types, in practice very similar to Java interfaces except that struct's subtyping is, of course, structural. Whiteoak also has some features for post-hoc object extension, and a form of trait composition. Unlike Unity, Whiteoak has been implemented, and a type checking algorithm is described, although the type system has not been formalized.

Beginning in Scala 2.6, Scala supports structural types as refinements of the top type `AnyRef` [36]. Structural types may appear wherever Scala types are expected, and generally may take part in Scala's rich and multifaceted type system. The formalizations of Scala's type systems, νObj [37], FS_{alg} [21], and μDOT [3] are nominal, and do not incorporate

structural types (Scala's "refinements"). Philosophically, Scala, Whiteoak, and Unity share a single approach: adding structural types to an existing nominal system. Our approach is the opposite.

Brands and structural typing have also been used in more practical languages. Trellis/OWL was based on structural types [42] although without brands. Strongtalk [14] is an optional (AKA pluggable) type system for Smalltalk: in the original version of Strongtalk, the types were structural with optional brands, again very similar to our design, although a later version of Stongtalk abandoned brands and adopted declared subtyping and matching relationships [12]. Diamondback Ruby adopted a type system very similar to Strongtalk's to type check Ruby programs [23]. Diamondback Ruby does not use brands, although it employs both nominal types generated from classes, and structural types to describe individual objects. Many other efforts to add types to dynamic languages, such as Typed Scheme/Racket, provide a set of type combinators such as intersection or union which allow the build up of more complicated type aliases [43]. Typed Racket in particular takes advantage of the language's underlying extensibility to include the type system as a library, rather than in the language.

Trademarks have been proposed for ECMAScript 6 [28], which provide a very similar model of branding for the language. Trademarks are also split between a branding object and a 'guard', the latter of which is conceptually a type, with the same mechanism of hiding the branding object while exposing the guard to prevent fraudulent branding. As a dynamically-typed language, combining static reasoning about trademarks with a static structural type system would be useful.

The implementation of brand objects, with the brand type accessible through a method on the brand itself, is reminiscent of the path dependent types in μDOT [3]. Following a chain of statically-known values is required in order to resolve the type statically, and is part of the requirement for the **let** construct. μDOT focuses more on strictly defined associated types, whereas Grace currently does not allow type literals to include other type declarations inside of themselves.

Recent work includes the Tagged Objects of Lee et al. in Wyvern, which adds nominal tags on top of an existing structural type system [30]. This approach focuses on the type theory of tags, and provides new primitive type and matching constructs as an extension to the language, with new static typing rules. This differs from our branding, which introduces the nominal types through existing language features including dialects and runtime reflection.

8 Conclusion

In this paper, we have described how we added nominal types (via brand objects) as a minimal extension to Grace's structural type system. We have demonstrated this extension on top of the existing implementation Hopper, drawing on Grace's pattern matching and dialects. We have provided several case studies of brands in the existing implementation, and have modeled the new type system and proved it sound. The key advantage of our approach is that brands are a much smaller addition to a preexisting structural type system than structural types are to a preexisting nominal type system, and required only a minimal change to the underlying language. So, if you are going to design a language that combines nominal and structural typing, our strong advice is to follow Modula-3: start with a structural system and then add nominal types, rather than to follow Unity, Whiteoak and Scala, which start with a nominal system and then add what amounts to another entire (structural) type system alongside.

References

1. Martin Abadi. Baby Modula-3 and a theory of objects. *Journal of Functional Programming*, 4(2):249–283, 1994.
2. Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.
3. Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *OOPSLA*, 2014.
4. Henk Barendregt. *The Lambda Calculus*. North-Holland, revised edition, 1984.
5. G. M. Birtwistle, O. J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Studentlitteratur, 1979.
6. Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. Grace: the absence of (inessential) difficulty. In *Onward!*, pages 85–98, 2012.
7. Andrew P. Black, Kim B. Bruce, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. Seeking Grace: a new object-oriented language for novices. In *SIGCSE*, pages 129–134, 2013.
8. Andrew P. Black, Kim B. Bruce, and James Noble. The Grace programming language (draft specification version 0.3.1303). `http://gracelang.org/documents/grace-spec031303.pdf`, 2013.
9. Andrew P. Black, Eric Jul, Norman Hutchinson, and Henry M. Levy. The development of the Emerald programming language. In *History of Programming Languages III*. ACM Press, 2007.
10. Andrew P. Black and Jens Palsberg. Foundations of object-oriented languages – workshop report. *SIGPLAN Notices*, 29(3):3–11, 1994.
11. John Tang Boyland. The problem of structural type tests in a gradual-typed language. In *FOOL*, New York, NY, USA, 2014. ACM.
12. Gilad Bracha. The Strongtalk type system for Smalltalk. In *OOPSLA Workshop on Extending the Smalltalk Language*, 1996.
13. Gilad Bracha. Pluggable Type Systems. OOPSLA workshop on revival of dynamic languages, 2004.
14. Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*, 1993.
15. Kim B. Bruce. A paradigmatic object-oriented programming language: design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.
16. Kim B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.
17. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.
18. Luca Cardelli, James E. Donahue, Mick J. Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *POPL*, pages 202–212, 1989.
19. William R. Cook. On understanding data abstraction, revisited. In *OOPSLA*, 2009.
20. William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 125–135, 1990.
21. Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for Scala type checking. In *MFCS*, pages 1–23, 2006.
22. Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.
23. M. Furr, J.-H. An, J. Foster, and M.J. Hicks. Static type inference for Ruby. In *SAC*, pages 1859–1866, 2009.
24. Joseph Gil and Itay Maman. Whiteoak: Introducing structural typing into Java. In *OOPSLA*, 2008.
25. Neal Glew. Type dispatch for named hierarchical types. In *ICFP*, New York, NY, USA, 1999. ACM.

26 Michael Homer, Timothy Jones, James Noble, Kim B. Bruce, and Andrew P. Black. Graceful dialects. In *ECOOP*, pages 131–156, 2014.

27 Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. Patterns as objects in Grace. In *DLS*, New York, NY, USA, 2012. ACM.

28 Waldemar Horwat and Mark Miller. ES6 Strawman: Trademarks. `http://wiki.ecmascript.org/doku.php?id=strawman:trademarks`, 2011.

29 Timothy Jones and James Noble. Tinygrace: A simple, safe and structurally typed language. In *FTFJP*. ACM, New York, NY, USA, 2014.

30 Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. A theory of tagged objects. In *ECOOP*, 2015.

31 Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 3.12 documentation and user's manual, 2011.

32 Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *ECOOP*, 2008.

33 Donna Malayeri and Jonathan Aldrich. Is structural subtyping useful? An empirical study. In *ESOP*, 2009.

34 John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *TOPLAS*, 10(3), 1988.

35 Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

36 Martin Odersky. *The Scala Language Specification: Version 2.9*. Programming Methods Laboratory, EPFL, Switzerland, 2011.

37 Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP*, pages 201–224, 2003.

38 Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, Chichester, 1994.

39 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

40 Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4:207–247, 1994.

41 Rob Pike and The Go Team. The Go programming language specification. `http://golang.org/ref/spec`, 2014.

42 Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/OWL. In *OOPSLA*, 1986.

43 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *POPL*, 2008.

44 Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregt's variable convention in rule inductions. In *CADE*, pages 35–50, 2007.

Access-rights Analysis in the Presence of Subjects

Paolina Centonze[1], Marco Pistoia[2], and Omer Tripp[2]

1 **Iona College**
New Rochelle, New York, USA
pcentonze@iona.edu

2 **IBM T. J. Watson Research Center**
Yorktown Heights, New York, USA
{pistoia,otripp}@us.ibm.com

Abstract

Modern software development and run-time environments, such as Java and the Microsoft .NET Common Language Runtime (CLR), have adopted a declarative form of access control. Permissions are granted to code providers, and during execution, the platform verifies compatibility between the permissions required by a security-sensitive operation and those granted to the executing code. While convenient, configuring the access-control policy of a program is not easy. If a code component is not granted sufficient permissions, authorization failures may occur. Thus, security administrators tend to define overly permissive policies, which violate the Principle of Least Privilege (PLP).

A considerable body of research has been devoted to building program-analysis tools for computing the optimal policy for a program. However, Java and the CLR also allow executing code under the authority of a *subject* (user or service), and no program-analysis solution has addressed the challenges of determining the policy of a program in the presence of subjects.

This paper introduces Subject Access Rights Analysis (SARA), a novel analysis algorithm for statically computing the permissions required by subjects at run time. We have applied SARA to 348 libraries in IBM WebSphere Application Server – a commercial enterprise application server written in Java that consists of >2 million lines of code and is required to support the Java permission- and subject-based security model. SARA detected 263 PLP violations, 219 cases of policies with missing permissions, and 29 bugs that led code to be unnecessarily executed under the authority of a subject. SARA corrected all these vulnerabilities automatically, and additionally synthesized fresh policies for all the libraries, with a false-positive rate of 5% and an average running time of 103 seconds per library. SARA also implements mechanisms for mitigating the risk of false negatives due to reflection and native code; according to a thorough result evaluation based on testing, no false negative was detected. SARA enabled IBM WebSphere Application Server to receive the Common Criteria for Information Technology Security Evaluation Assurance Level 4 certification.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages, D.4.6 Security and Protection

Keywords and phrases Static Analysis, Security, Access Control

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.222

1 Introduction

Modern software development and run-time environments, such as Java and the Microsoft .NET CLR, have adopted a form of declarative access control. Developers do not have to encode access-control-policy definition and enforcement capabilities inside their applications on a case-by-case basis because these capabilities are integrated within the underlying

© Paolina Centonze, Marco Pistoia, and Omer Tripp;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 222–246

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

platforms. Given the highly distributed nature of today's applications, and the fact that code can be dynamically loaded, it is essential to restrict resource access based on code providers. At run time, when a security-sensitive resource is about to be accessed, the access-control enforcement mechanism verifies that all the code responsible for that resource access is sufficiently trusted.

Decoupling access-control definition and enforcement from the application code promotes code portability and reusability, and minimizes the risk of security holes. Nevertheless, configuring the access-control policy of an application can be complicated. A security administrator must be informed of all the security-sensitive operations that an application may attempt to perform at run time and grant the program components all the permissions necessary to complete those operations. If some of the necessary permissions are not granted, run-time authorization failures will occur. On the other hand, granting unnecessary permissions would constitute a violation of a fundamental security rule, known as the Principle of Least Privilege (PLP) [33]. Therefore, it is essential that exactly the permissions necessary for the program to execute without authorization failures be granted.

1.1 Existing Approaches

For these complications, researchers have studied extensively how program analysis can be used to automatically infer an *optimal* access-control policy: one that is neither too restrictive nor too permissive. Numerous approaches have been proposed to address this problem in Java and CLR [31, 21, 7, 6, 36, 12, 29, 8, 38, 10, 37, 11, 13, 23]. More recently, permission analysis has been extended to Android [14, 4]. Hybrid techniques have also been attempted. For example, in order to better disambiguate the resources guarded by the various permissions and the mode in which those resources are accessed, static permission analysis has been integrated with string analysis [16], and in order to mitigate the false positives arising during static analysis and the false negatives typical of dynamic analysis, a hybrid static/dynamic approach has been studied [9]. Solutions have also been proposed that simultaneously integrate access-control and information-flow enforcement [5, 1, 15, 30].

What is critically missing from all existing approaches is treatment of *subjects*. These are users or services that map to one or more identities, called *principals*, each of which can be granted permissions. The concept of subject exists in Java and the CLR. Henceforth, we focus our discussion on Java for space and readability. In Java, the security component responsible for subject-based access control is the Java Authentication and Authorization Service (JAAS) [24]. Developers can invoke specific JAAS APIs to cause parts of a program to be executed under the authority of a subject. However, it is burdensome to determine manually the access-control policy of a program in the presence of subject-executed code. This amounts to resolving the principals associated with each subject, the permissions granted to each subject as a result of such associations, and the portions of the program executed under the authority of each individual subject.

1.2 This Paper

The work presented in this paper was motivated by the requirement to port IBM WebSphere Application Server[1] to the Java permission- and subject-based security model. While performing this complex task, we discovered severe access-control violations. Of the 348

[1] `http://ibm.com/software/products/appserv-was`

libraries comprising the application server, totaling over 2 million lines of code (MLOC), only 102 libraries had an associated policy with subject permissions in place. Those policies, which had been defined based on manual code inspection and unit testing, exhibited numerous errors. The remaining 246 policies did not account for subjects at all. Furthermore, developers had often used the subject APIs incorrectly, thereby introducing bugs in the code that could not be fixed by simply modifying a security policy.

In this paper, we show how we filled the gap between the need for a subject-sensitive access-rights analysis and the lack of an automated analysis framework to achieve this. We have formulated Subject Access Rights Analysis (SARA), the first comprehensive framework for access-rights analysis, whose purpose is to automatically determine which methods in the program under analysis are executed under the authority of a given subject at run time.

The input to SARA is the object code of a program. SARA statically builds a specialized representation of the program in the form of a context-sensitive call graph [32], so as to capture precisely run-time permission requirements. SARA annotates the call graph with subject-granted access rights and permission requirements, and uses this information to determine the subject-granted permissions under which a method will be executed at run time. This information is used to decide which access rights should be granted to subjects and code to prevent run-time authorization failures, and which access rights should be revoked to prevent PLP violations.

When applied to IBM WebSphere Application Server, SARA was able to synthesize access-control policies for the 105 libraries previously missing a specification. SARA was also able to detect all the errors we discovered manually in existing policies authored by seasoned developers and system administrators, and automatically revised the respective policies to eliminate all the errors. When given as input the 348 libraries of IBM WebSphere Application Server, SARA detected 263 PLP violations, 219 cases of policies with missing permissions, and 29 bugs that led code to be unnecessarily executed under the authority of a subject. SARA corrected all these vulnerabilities automatically, and additionally synthesized fresh policies for all the libraries, with a false-positive rate of 5% and an average running time of just over 100 seconds per library. SARA implements several mechanisms for mitigating the threat of false negatives due to the presence of native code or reflective method calls in the code under analysis. Based on a thorough evaluation of the results, no false negative was detected. SARA enabled IBM WebSphere Application Server to receive the Common Criteria for Information Technology Security Evaluation Assurance Level (CC EAL) 4 certification.[2]

2 Technical Background

This section describes how the Java and CLR stack-inspection mechanism works, focusing on Java as a reference.

2.1 Basic Concepts

In order to prevent confused-deputy attacks [20], when access to a restricted resource is attempted, all code currently on the call stack must be authorized to access that resource. In Java, the `SecurityManager`, if active, triggers access-control enforcement by invoking method `checkPermission` in class `AccessController`. This static method takes a `Permission` object `p` as a parameter and performs a stack walk backwards to verify that every caller in the

[2] `http://commoncriteriaportal.org/`

current thread of execution has been granted the access right represented by `p`. If that is not the case, a `SecurityException` is thrown.

Though programmatic security is possible, access rights are preferably granted declaratively, in a policy database external to the application code. This enhances code portability and reusability. Access rights are by default denied to all code and subjects. Untrusted code and subjects will only be allowed to perform basic operations that cannot harm the system. For a restricted operation to succeed, all the code on the thread's stack must be explicitly granted the right to execute that operation.

Access rights are represented as objects of type `Permission`. Each `Permission` type must be a subclass of the `Permission` abstract class. When a `Permission` object is constructed, it can take zero, one, or two `String` objects as parameters. If present, the first parameter is called the *target* of the `Permission` object and represents the resource being protected; the second parameter is called the *action* of the `Permission` object and represents the mode of access. The target and action are used to better qualify the resource guarded by the `Permission` object. For example, the following line of code can be used to construct a `Permission` object representing the right to access the `log.txt` file in read/write mode:

```
Permission p = new FilePermission("log.txt", "read,write");
```

Given a `Permission` object `p`, `p`'s fully-qualified `Permission` class name along with `p`'s target and action, if any, uniquely identify the authorization requirement represented by `p`. Therefore, for authorization purposes, a `Permission` object `p` can be characterized solely based on `p`'s *permission identifier*, which consists of `p`'s fully-qualified class name and the values of the `String` instances used to instantiate `p`. In fact, authorizations are granted to programs and principals by simply listing the corresponding permission identifiers in a flat-file policy database dubbed the *policy file*.

The `Permission` class contains an abstract method, `implies`, which itself takes a `Permission` object as a parameter and returns a `boolean` value. Every non-abstract `Permission` subclass must implement `implies`. If `p` and `q` are two objects of type `Permission` such that `p.implies(q)` returns `true`, then this means that granting `p` implicitly grants `q` as well. For example, `p` could be the `Permission` object constructed above and `q` could be the Permission object constructed with the following line of code:

```
Permission q = new FilePermission("log.txt", "write");
```

If `p` is an instance of `AllPermission`, then `p.implies(q)` returns `true` for any `Permission` object `q`.

Access rights may be granted to code based on the *code source*, which is a combination of the *code base* – the network location from which the code is coming – and the certificates of the entities that digitally signed the code. Access rights are granted to classes. At run time, each class is loaded by a class loader. When it loads a class, a class loader constructs a *protection domain* characterizing the origin of the class being loaded, and associates it with the class itself. A protection domain, which is represented as an object of type `ProtectionDomain`, encapsulates the class' code source (represented as a `CodeSource` object) and an object of type `PermissionCollection` containing all the `Permission` objects corresponding to the access rights granted to that code source. When invoked with an argument `p` of type `Permission`, method `checkPermission` performs a stack traversal backwards, and verifies that each of the `ProtectionDomains` in the current thread of execution contains at least one `Permission` that implies `p`. The set of `Permissions` effectively granted to a thread of execution is, therefore, the intersection of the sets of `Permissions` implied by all the `ProtectionDomains` associated with the stack.

2.2 Subjects-based Authorization

Authorization decisions can also be based on the subject executing the code. In Java, a subject is represented as an object of type `Subject`. When a subject is authenticated, the corresponding `Subject` instance is populated with associated identities called *principals*, represented as objects of type `Principal`. A subject may have multiple associated principals. For example, if the subject is a person, two of that subject's principals could be that person's name and social security number, depending on which means the person used for authentication.

Access rights are granted to code and principals, though not directly to subjects. The set of access rights granted to a subject is the union of the sets of access rights granted to the subject's authenticated principals. The `Subject` class exposes static methods `doAs` and `doAsPrivileged` to perform a restricted operation with the access rights granted to a subject.

- `doAs` accepts two parameters: The first is a `Subject` object, and the second is either a `PrivilegedAction` or `PrivilegedExceptionAction` object `o`. The code in the `run` method of argument `o` is executed with the intersection of the sets of `Permissions` granted to the code on the call stack. However, `doAs` adds the `Permissions` granted to the subject's principals to the stack frames following the call to `doAs`.
- `doAsPrivileged` is similar to `doAs`, but accepts an additional third parameter: an `AccessControlContext` object. This object encapsulates an array of `ProtectionDomain` objects. Just like `doAs`, `doAsPrivileged` also adds the `Permission` objects granted to the subject's principals to the subsequent stack frames. However, unlike the case of `doAs`, when the `checkPermission` method is called with a `Permission` parameter `p` after a call to `doAsPrivileged`, the predecessors of `doAsPrivileged` are not required to exhibit a `Permission` that implies `p`. Rather, `doAsPrivileged` demands the `ProtectionDomains` that are embedded in the `AccessControlContext` passed to it as a parameter to exhibit such a `Permission`.

A library can execute a security-sensitive operation without propagating the corresponding permission requirements to its clients. This is done by wrapping that operation into the `run` method of a `PrivilegedAction` or `PrivilegedExceptionAction` object `o`, and then by calling `AccessController.doPrivileged` with `o` as a parameter. As a motivation for this, consider the case of a library that has been designed to open socket connections on behalf of its clients. For any such socket connection, it is justified to demand that both the library and the client be granted the relevant `SocketPermission`. However, if that library additionally logs to a file the details of the connections that it opens, then it would not be appropriate to demand the `FilePermission` of the client. Only the library should be demanded that. If a malicious client were granted that `Permission`, then that client could compromise the integrity of the log file.

We define subject-executed code as *unnecessary* if it does not lead to any call to `checkPermission`, and as *redundant* if it leads to a call to `checkPermission` only through a `doPrivileged` call. Our objective, accomplished by SARA, is to detect subject-executed code that is unnecessary or redundant. Not only can such code constitute a PLP violation, but it can further negatively affect the performance of the program. SARA also detects and automatically corrects policies that are overly restrictive or overly permissive. An overly restrictive policy is one that does not grant the application code or the subjects executing it sufficient rights to meet the security requirements of the application, in which case run-time authorization failures may occur, causing the application to crash. An overly permissive policy grants subjects or code unnecessary permissions, which constitutes a PLP violation.

3 Technical Overview

In this section, we provide a high-level summary of the technical description appearing in the next two sections. We survey the flow and milestones of the SARA framework and highlight its main technical novelties.

3.1 Call-graph Representation

The first step of SARA is to model the behavior of the program `P` in the form of a call graph. The call-graph representation conveys the global control flow of the program. This is done by iteratively computing the structure of calls within `P` starting from the entry-point methods (*i.e.*, those that are externally invocable). Since some (in reality, most) of the call sites are virtual, requiring resolution of the receiver to disambiguate the target method(s), call-graph construction is interleaved with pointer analysis [18]. While other call-graph construction techniques exist [35], combining pointer analysis into call-graph construction boosts precision and makes points-to information available to downstream analyses, which we indeed utilize in SARA.

In the iterative process, call-graph construction resolves call sites into target methods, and meanwhile, pointer analysis resolves variables, array elements and object fields into abstractions of run-time objects. Specifically, each object is abstracted as its allocation site, which ensures finitely many object abstractions. Resolution of call sites may reveal more allocation sites to be tracked by the pointer analysis. At the same time, disambiguation of virtual call sites based on the points-to image of the receiver variable may enable resolution of more call sites. This process is iterated to fixpoint.

3.2 Permission Hierarchy

The next step is to model the partial order between `Permission` identifiers according to the `implies` relation, introduced in Section 2.1. Doing so purely statically is a serious challenge. `implies` is typically implemented as a series of nontrivial tests on its `Permission` argument, which are hard to track accurately in a static manner. At the same time, preciseness is crucial when modeling the permission structure, as this is the foundation underlying all the analysis steps to follow.

Given these considerations, we have adopted a novel strategy in SARA of allocating `Permission` instances dynamically and invoking their `implies` method in a sandboxed environment. For this, SARA first traverses the call graph to retrieve all the allocation sites of `Permission` subclasses. Then, for each allocation site in turn, a string analysis is applied to resolve constructor arguments into concrete values, as explained in Section 6.2. Assuming for now that the resolution is successful, SARA can recover the hierarchical relationship between `Permission` instances precisely by instantiating the instances and invoking `implies` in a sandboxed environment over all instance pairs. Sandboxing is achieved by activating the `SecurityManager`, and at the same time depriving the `Permission` receiver object of any permission. In this way, `implies` is prevented from performing any security-sensitive operation.

For cases where the string analysis fails, in part or in whole, we fall back on a per-permission specification of conservative argument approximations. As an example, inability to resolve the file pattern specified as the first argument of `FilePermission` is resolved conservatively as `"<<ALL_FILES>>"`, a notation used in Java to symbolize all the possible files in the file system. In situation where only a substring can be disambiguated, SARA resolves the file

name partially. For example, if the name of a file guarded by a `FilePermission` object is obtained by concatenating `String` constant `"C:/"` with the dynamic value of `String` variable `x`, SARA computes the file name as `"C:/*"`, which is more precise than `"<<ALL_FILES>>"`. Similarly, inability to resolve the second argument, denoting the access mode, is resolved as `"read,write,execute,delete"`.

For application-specific `Permission` subclasses, the user of SARA has the option to extend the specification. If this is not done and the string analysis is unable to resolve a `String` value, we conservatively resolve the `Permission` as `AllPermission`. We comment, from our experience, that we rarely encountered failures by the string analysis.

3.3 Access-rights Annotations

Having a model of the program's calling structure and the relationships between the various `Permission` objects, SARA identifies, within the call graph, invocations of the access-rights-related APIs: namely `checkPermission` and `doPrivileged`. At a given call site invoking `checkPermission`, SARA retrieves a conservative approximation of the checked `Permissions` as the points-to set of the argument.

SARA then statically simulates the run-time stack inspection mechanism by propagating the requirement to possess the permission(s) arising at `checkPermission` backwards to all the transitive callers of `checkPermission`. As in the concrete semantics, backward propagation is aborted at `doPrivileged` callers. At this stage, every call-graph node is mapped to a set of required `Permissions`. The naive solution, as formulated by existing approaches, is to stop the analysis at this point and missing permissions to relevant code to ensure normal execution. However, this ignores the presence of `Subjects`, as we next explain.

3.4 Subject-specific Annotations

To account for `Subjects`, SARA next traverses the call graph in order to identify `doAs` and `doAsPrivileged` calls, wherein the first parameter is of type `Subject`. Again thanks to points-to information, that parameter is resolved into one or more abstract `Subject` instances. SARA then consults the points-to graph per each of the instances. In particular, the `Subject` class has a field `principals` of type `Set` that stores all the corresponding principals of a subject. As discussed in Section 2.1, it is the `Principals` rather than the `Subject` that are granted `Permissions`, and so SARA uses the points-to graph to compute (i) the set of `Principal` object abstractions pointed to by the `principals` field and (ii) their respective constructor values via constant propagation. Next, `Permissions` are extracted from the policy based on the *principal identifiers*, consisting of concrete `Principal` subtypes and initialization arguments. The respective `Permissions` of a particular `Subject` object flowing into a `checkPermission` call are the union of all the `Permissions` granted to its corresponding `Principals`, as determined conservatively via the points-to information.

With this information, SARA revisits the annotations computed in the previous step. The goal is to relax the demand for missing `Permissions` if these are provided already under the authority of a `Subject`. This is done as follows: At a given `doAs` or `doAsPrivileged` invocation point, SARA resolves the potential `Subjects` via the points-to map. Because the points-to information is conservative, the `Permissions` *guaranteed* to be provided at that point are the intersection, rather than union, of the `Permissions` held by the different `Subjects`. At this point, the `Permissions` shared by all `Subjects` are propagated forward and unioned with the `Permissions` that the given call-graph nodes already have.

This additional analysis step is significant: Naively the code may appear to be missing `Permissions`, thereby soliciting the system administrator to grant it undue access rights. Those can then be abused either by the code itself or by a malicious user. As an illustration, assume a text editor that requires a special `Permission` to edit a specific document. The subject owning that document possesses that `Permission`. Ignoring the subject, however, would lead the analysis tool to a recommendation that the code itself should own that `Permission`, effectively enabling other users to access the document.

3.5 Error Analysis

The final step is to analyze the artifacts computed by SARA and apply corresponding corrections or synthesis. The first scenario is detection of missing `Permissions` under consideration of `Subjects`. The solution, similarly to existing approaches, is to grant the missing `Permissions`. However, unlike existing approaches, SARA grants the `Permissions` by default to the subject rather than to the code. This is the more conservative policy, as granting `Permissions` to the code enables the respective operations per all subjects as well as the code itself. This is, however, configurable if the user wishes to override the default policy.

The second scenario arises when there is duplicity across the `Permissions` granted to the code and subject. That is, the code already possesses a needed `Permission`, but there is also a call to `doAs` or `doAsPrivileged` to enable that same `Permission` via the `Subject`. This form of redundancy constitutes a PLP violation. For the same rationale explained above, by default SARA revokes the `Permission` from the code rather than the subject, though again this is a configurable choice.

4 Static-analysis Framework

The static-analysis framework presented in this paper uses graph theory to represent the execution of a program and lattice theory to model the flow of information in the program. A program is modeled as a call graph and its associated points-to graph. A *call graph* is a directed graph $G = (N, E)$, in which nodes correspond to method invocations. Two nodes $n_1, n_2 \in N$ are connected by an edge $(n_1, n_2) \in E$ iff the analysis conservatively establishes that the method represented by n_1 may invoke the method represented by n_2 at run time [18]. A *points-to graph* is a directed bipartite graph in which each vertex corresponds to either a program variable or an object abstraction, and an edge indicates a points-to relation [3]. Both the call graph and the points-to graph presented in this paper are tailored to authorization analysis.

4.1 Permission Abstraction

When designing an algorithm that computes data flow over a graph, for Tarski's theorem to guarantee convergence to a fixed point in polynomial time, it is important to make sure that (i) the data-flow functions defined at each node map a lattice into itself, (ii) the lattice is complete and has finite height, and (iii) the data-flow functions are monotonic with respect to the lattice's partial order [17]. Although in Section 3 we intuitively talked about partial order between `Permissions` as well as unions and intersections of `Permission` sets, a more precise discussion is needed in order to formalize the analysis and guarantee its convergence. This section defines the lattice used as domain and codomain for the data-flow functions employed by SARA.

4.1.1 Permission Graph

The relationships induced by the `implies` methods among the `Permission` objects associated with a program under analysis can be modeled using a *permission graph* $H = (P, F)$, where P is a set of `Permission` objects and $F \subseteq P \times P$ is a set of edges of the form $\mathtt{p} \rightarrow \mathtt{q}$, where $\mathtt{p}, \mathtt{q} \in P$. Building the permission graph during the analysis requires detecting all the `Permission` objects that appear in the program, since each `Permission` object corresponds to an element of P. To build F, each $\mathtt{p} \in P$ must be explicitly instantiated. This operation is done during the analysis using reflection based on `p`'s permission identifier. Next, for each pair $(\mathtt{p}, \mathtt{q}) \in P \times P$, SARA runs `p.implies(q)` to decide whether $\mathtt{p} \rightarrow \mathtt{q} \in F$ or not.

4.1.2 Permission Lattice

The `implies` method does not necessarily induce a partial order on P. For instance, if `p` and `q` are two different `Permission` objects with the same permission identifier, then $\mathtt{p} \rightarrow \mathtt{q}$ and $\mathtt{q} \rightarrow \mathtt{p}$. Therefore, H does not have the structure of a lattice, which would be desirable when performing data-flow analysis. However, H can be transformed into a lattice as follows. Given $\mathtt{p}, \mathtt{q} \in P$, let $\mathtt{p} \overset{*}{\rightarrow} \mathtt{q}$ denote the presence of a path from `p` to `q` in H. `Permission` objects $\mathtt{p}, \mathtt{q} \in P$ belong to the same Strongly Connected Component (SCC) of H iff $\mathtt{p} \overset{*}{\rightarrow} \mathtt{q}$ and $\mathtt{q} \overset{*}{\rightarrow} \mathtt{p}$. In this case, `p` and `q` are said to be *equivalent*, denoted by $\mathtt{p} \equiv \mathtt{q}$. Let $H_{\equiv} = (P_{\equiv}, F_{\equiv}) = (P^{SCC}, F^{SCC})$ be the Directed Acyclic Graph (DAG) induced by collapsing each SCC of H into a single node. It is easy to see that $H_{\equiv}$ has the structure of a partially ordered set.

$H_{\equiv}$ can be given the structure of a lattice by adding two new elements to $P_{\equiv}$: $\top = \mathit{AllPermission}$, denoting the SCC containing all instances of the `AllPermission` class (if such an SCC does not already exist), and $\bot$, which denotes absence of authorizations. As we have already observed, an instance of `AllPermission` implies any other `Permission`. Therefore, $\forall p \in P_{\equiv}.\ \top \rightarrow p$. Additionally, we impose that $\forall p \in P.\ p \rightarrow \bot$. Let $\overline{P_{\equiv}}$ be the augmented set $P_{\equiv} \cup \{\top, \bot\}$, and let $\overline{F_{\equiv}}$ be the superset of $F_{\equiv}$ obtained by adding into $F_{\equiv}$ the edges involving the new elements $\top$ and $\bot$. It is easy to prove that the graph $\overline{H_{\equiv}} = (\overline{P_{\equiv}}, \overline{F_{\equiv}})$ represents the lattice $(\overline{P_{\equiv}}, \sqsupseteq)$, where $\sqsupseteq$ is the partial-order relation defined on a subset of $\overline{P_{\equiv}} \times \overline{P_{\equiv}}$ by $p \sqsupseteq q \Longleftrightarrow p \overset{*}{\rightarrow} q, \forall p, q \in \overline{P_{\equiv}}$. This lattice is called the *permission lattice*. Its top and bottom elements are $\top$ and $\bot$, respectively. The *meet* and *join* operations $\sqcap, \sqcup \colon \overline{P_{\equiv}} \times \overline{P_{\equiv}} \rightarrow \overline{P_{\equiv}}$, induced by $\sqsupseteq$ on $\overline{P_{\equiv}}$, are defined as follows, respectively:

1. $p \sqcap q = r$, where $r \in \overline{P_{\equiv}}$ is such that $p, q \sqsupseteq r \wedge r \sqsupseteq x, \forall x \in \{\overline{P_{\equiv}} : p, q \sqsupseteq x\}$
2. $p \sqcup q = r$, where $r \in \overline{P_{\equiv}}$ is such that $r \sqsupseteq p, q \wedge x \sqsupseteq r, \forall x \in \{\overline{P_{\equiv}} : x \sqsupseteq p, q\}$

4.1.3 Permission-set Lattice

The authorization analysis performed by SARA is modeled as a data-flow problem wherein *sets* of access rights (rather than *single* access-right elements) are propagated through the call graph. The analysis performs meet and join operations on those sets. Therefore, it is necessary to lift the meet and join operations defined in Section 4.1.2 over elements of $\overline{P_{\equiv}}$ to sets of elements. In other words, it is necessary to define a lattice structure over the powerset $\mathcal{P}(\overline{P_{\equiv}})$.

Naturally, $\mathcal{P}(\overline{P_{\equiv}})$ has the lattice structure defined by regular set inclusion, $\supseteq$, which induces set intersection, $\cap$, and set union, $\cup$, as meet and join operations. However, the lattice $(\mathcal{P}(\overline{P_{\equiv}}), \cap, \cup)$ would not be appropriate for authorization analysis. For example, let `p` and `q` be two `Permission` objects instantiated through the two following lines of code:

```
Permission p = new FilePermission("C:\\*", "read");
Permission q = new FilePermission("C:\\log.txt", "read,write");
```

Following Section 4.1.2, let p and q represent their respective SCCs in the permission graph. SARA forward propagates elements of $\mathcal{P}(\overline{P_{\equiv}})$ across the call graph, where the operation applied to the lattice elements must be meet. If n is a node in the call graph, n_1 and n_2 are two predecessors of n, and SARA has established that n_1 and n_2 will be potentially executed with access rights represented by sets $\{p\}$ and $\{q\}$, respectively, then the safest assumption if the $(\mathcal{P}(\overline{P_{\equiv}}), \cap, \cup)$ lattice structure were adopted would be that n will be executed with an empty set of access rights since $\{p\} \cap \{q\} = \varnothing$. This result is overly conservative. Instead, let `r` be any `FilePermission` object instantiated through a line of code similar to the following:

```
Permission r = new FilePermission("C:\\log.txt", "read");
```

Since $p \sqcap q = r$, it is more desirable to conclude that n will be executed with access-right set $\{r\}$. Conversely, computing the permission requirements induced by the stack inspection mechanism requires backward propagation of $\mathcal{P}(\overline{P_{\equiv}})$ elements across the call graph. In such cases, the operation performed on the lattice elements must be join. If the set union operation, $\cup$, were applied to sets $\{p\}$ and $\{q\}$, then the result would be $\{p, q\}$. However, $p \sqcup q = v$, where v is the SCC of any `FilePermission` object `v` instantiated through a line of code similar to the following:

```
Permission v = new FilePermission("C:\\*", "read,write");
```

Therefore, it is more desirable to have a join operation on $\mathcal{P}(\overline{P_{\equiv}})$ that, once applied to $\{p\}$ and $\{q\}$, gives v as a result.

A more meaningful lattice structure can be given to $\mathcal{P}(\overline{P_{\equiv}})$ based on the lattice structure of $(\overline{P_{\equiv}}, \sqsupseteq)$. A set $Q \in \mathcal{P}(\overline{P_{\equiv}})$ is defined as *canonical* iff

$$\forall p, q \in Q.\ p \neq q \Longrightarrow (p \not\sqsupseteq q) \wedge (q \not\sqsupseteq p)$$

Intuitively, if $Q \in \mathcal{P}(\overline{P_{\equiv}})$ is canonical, then no element in Q implies any other element in Q except itself. A *canonical reduction* function $\chi : \mathcal{P}(\overline{P_{\equiv}}) \to \mathcal{P}(\overline{P_{\equiv}})$ can be introduced that maps any set $Q \in \mathcal{P}(\overline{P_{\equiv}})$ to its subset $\chi(Q) \in \mathcal{P}(\overline{P_{\equiv}})$ obtained by removing from Q all the elements that are implied by some other element of Q. Formally, $\chi(Q) = \{q \in Q : \forall r \in Q.\ r \not\sqsupseteq q\}$. The χ function is well defined because for any $Q \in \mathcal{P}(\overline{P_{\equiv}})$, there exists one and only one canonical set corresponding to Q [28]. Functions $\sqcap, \sqcup : \mathcal{P}(\overline{P_{\equiv}}) \times \mathcal{P}(\overline{P_{\equiv}}) \to \mathcal{P}(\overline{P_{\equiv}})$ are defined, respectively, as follows:

1. $\forall Q, R \in \mathcal{P}(\overline{P_{\equiv}}).\ Q \sqcap R = \chi(\{q \sqcap r : q \in Q, r \in R\})$
2. $\forall Q, R \in \mathcal{P}(\overline{P_{\equiv}}).\ Q \sqcup R = \chi(Q \cup R)$

It is easy to prove that both $\sqcap$ and $\sqcup$ are commutative, associative and mutually absorptive functions. Thus, $(\mathcal{P}(\overline{P_{\equiv}}), \sqcap, \sqcup)$ is a lattice, called the *permission-set lattice*, and $\sqcap$ and $\sqcup$ are its *meet* and *join* operations, respectively. For a given program or library, P is finite. This implies that the permission-set lattice associated with a program or library is also finite, and therefore complete. Its top element is the set $\{\mathit{AllPermission}\}$. Its bottom element is the empty set, $\varnothing$.

The partial order $\sqsupseteq$ induced by $\sqcap$ and $\sqcup$ on $\mathcal{P}(\overline{P_{\equiv}})$ is obtained as follows:

$$\forall Q, R \in \mathcal{P}(\overline{P_{\equiv}}).\ Q \sqsupseteq R \Longleftrightarrow Q \sqcap R = R$$

or, equivalently:

$$\forall Q, R \in \mathcal{P}(\overline{P_{\equiv}}).\ Q \sqsupseteq R \Longleftrightarrow Q \sqcup R = Q$$

Since the permission-set lattice is finite, its height, $\mathcal{H}(\mathcal{P}(\overline{P_{\equiv}}))$, is finite too. Specifically, $\mathcal{H}(\mathcal{P}(\overline{P_{\equiv}})) \in \mathcal{O}(|P|)$. Finally, the *difference operator*, $- : \mathcal{P}(\overline{P_{\equiv}}) \times \mathcal{P}(\overline{P_{\equiv}}) \to \mathcal{P}(\overline{P_{\equiv}})$, is obtained as follows:

$$\forall Q, R \in \mathcal{P}(\overline{P_{\equiv}}).\ Q - R = \chi(\{q \in Q : \nexists r \in R, r \sqsupseteq q\})$$

4.2 Permission-specific Call Graph

The first step in the analysis is to construct an augmented call graph, which we refer to as a *Permission-specific Call Graph* (PCG). SARA's call-graph and pointer-analysis implementation is based on the Watson Libraries for Analysis (WALA) static-analysis framework,[3] which allows for analysis of Java bytecode. A PCG is a directed multigraph $G = (N, E)$, where N is a set of nodes and E is a set of edges.

A node $n \in N$ represents a context-sensitive method invocation, and is uniquely identifiable via its *calling context*, which consists of (i) the concrete target method and (ii) the receiver and parameter values. n carries the following state: (i) the target method, (ii) object abstractions representing the receiver and parameters (in SARA, these are the concrete objects' allocation sites), and (iii) object abstractions representing the return value of the method. Therefore, a PCG is *context-sensitive* [32], because it uniquely distinguishes different invocations to the same methods by the calling context, with a context-sensitivity policy similar to Agesen's Cartesian Product Algorithm (CPA) [2]. With this policy, if a given method in a program is invoked twice, each time with different parameters according to the analysis abstraction, the PCG will model these two invocations as two distinct nodes.

An edge $e = m \xrightarrow{c} n \in E$ carries a label c denoting the call site through which the method `m` at m invokes the method `n` at n. Hence, G is a multigraph because `m` can invoke `n` multiple times, each time at a different call site. In the remainder of this paper, however, we will omit the label of an edge e, ad will simply write $e = (m, n)$, unless it is necessary to distinguish between different calls to `n` made by `m`.

The PCG has a unique node, $\overline{n} \in N$, called the PCG *root*. It represents external invocation of the entry points of the program under analysis. A node $n \in N$ is constructed iff it has been statically established that the method represented by n can be invoked during execution of the program under analysis. This guarantees that PCG nodes are reachable from $\overline{n}$. By inference, the PCG is a connected multi-graph.

We utilize Control-Flow Analysis (CFA) [27] during PCG construction to disambiguate heap objects according to their allocation sites. In addition to the CPA context sensitivity, the PCG construction process enforces the following properties:

- *Path insensitivity* [19], because it does not evaluate conditional statements and conservatively assumes that all branches are admissible
- *Intraprocedural flow sensitivity* [32], because it considers the order of execution of the instructions within each basic block, accounting for local-variable kills [22] and casting of object references
- *Interprocedural flow insensitivity* [32], because it makes the conservative assumption that all instance and static fields are subject to modification at any time, which may happen in the presence of other execution threads
- *Field sensitivity* [32], because an object's fields are represented distinctly in the solution (motivated, for example, by the `principals` field of a `Subject`)

[3] `http://wala.sf.net`

```
grant
 Principal javax.security.auth.x500.X500Principal
   "CN=John Doe, OU=Res, O=Org, C=US"
 Principal javax.security.auth.kerberos.KerberosPrincipal "jd@us.res.org" {
  permission java.util.PropertyPermission "user.home", "read";
  permission java.io.FilePermission "log.txt", "read";
  permission java.security.SecurityPermission "getPolicy";
};
```

Figure 1 Granting Access Rights to `Principal`s.

Among all the design choices listed above, the most important one is the CPA context sensitivity: While naively the call graph would represent each method as a single node, it is possible to impose several different representations of the same method inside a call graph per different contexts governing the execution of the method. SARA distinguishes between invocations of a method based on parameter values (including the receiver). The reason behind this choice is that access-rights-related information is largely encoded as parameter values.

For example, the `String` values flowing into a call to the `FilePermission` constructor determine which file(s) are guarded by the `FilePermission` object being constructed and what access modes are allowed. Therefore, in order to statically distinguish `FilePermission` objects protecting different files, it is necessary to disambiguate calls to the `FilePermission` constructor made with different arguments.

Similarly, the `Permission` parameter passed to the `checkPermission` method determines what access right will be demanded of all the callers on the stack or the subject associated with the current thread. Thus, from a security perspective, CPA context sensitivity is crucial since an authorization analysis needs to distinguish between different calls to `checkPermission` based on the `Permission` argument passed to it. Failure to do so would result in all the `checkPermission` calls in the program being represented as a single PCG node, which in turn would lead to conservatively joining together all the `Permission` requirements identified in the program.

5 The SARA Algorithm

This section describes the various steps of the SARA algorithm.

5.1 Permission-set Lattice Construction

According to the JAAS architecture, at run time, an authenticated `Subject` is granted the `Permissions` of the `Principals` encapsulated into it. `Permissions` in Java 2 are granted to a `Principal` based on two pieces of information:

1. The `Principal`'s fully qualified class name
2. The `Principal`'s *name*, which is the `String` value returned by the `getName` method of the `Principal` object

It is possible to include more than one `Principal` field in a `grant` statement, as in the policy file snippet of Figure 1. If a `grant` entry contains more than one `Principal` field, then the `Permissions` in the `grant` statement are awarded to the `Subject` associated with the current `AccessControlContext` only if that `Subject` contains *all* those specified `Principals`.

Let $\mathcal{D}$ be the security policy database associated with the program under analysis, and let $|\mathcal{D}|$ indicate the number of `grant` statements in $\mathcal{D}$. The first step of the SARA algorithm is to scan $\mathcal{D}$, retrieve from it the permission identifiers that are mapped to each `Principal` class and name pair, and use reflection to instantiate the `Permission` objects corresponding to those permission identifiers. Such `Permission` objects form the set P, which can be used to construct the permission graph $H = (P, F)$ and permission lattice $(\overline{P_\equiv}, \sqsupseteq)$, representing the access rights *granted* to the `Subject`s of a program based on their `Principal`s and the policy specified in $\mathcal{D}$.

It should be observed that it is not necessary to preconstruct the permission-set lattice $(\mathcal{P}(\overline{P_\equiv}), \sqcap, \sqcup)$; subsets of the permission lattice will be met at each node $n \in N$ every time an edge $e \in E$ of the form $e = (m, n) : m, n \in N$ is traversed during the fixed-point iteration described in Section 5.3. As will be explained in Section 5.3, in the worst case, the meet operation will have to be applied $\mathcal{H}(\mathcal{P}(\overline{P_\equiv}))$ times for each edge of the graph.

5.2 SARA Initialization

Let D be the subset of N consisting of all the nodes representing calls to `doAs` and `doAsPrivileged`. The first initialization step consists of computing D by simply iterating over N.

Both `doAs` and `doAsPrivileged` take a `Subject` object as one of the parameters. Let n be any element of D. SARA identifies the set $\psi_D(n) = \{\mathtt{s}_1, \mathtt{s}_2, \ldots, \mathtt{s}_j\} \subseteq S$ of all the possible `Subject` allocation sites that may have flowed to the formal `Subject` argument of the method call represented by n, where S is the union of all the `doAs` and `doAsPrivileged` `Subject` parameter sets in G. This defines a function $\psi_D : D \to \mathcal{P}(S)$ that maps nodes of D to subsets of S.

An authenticated `Subject` is granted all the `Permission`s of its associated `Principal`s. Therefore, for each `Subject` allocation site $\mathtt{s} \in S$, SARA identifies the set $\psi_S(\mathtt{s}) = \{\mathtt{u}_1, \mathtt{u}_2, \ldots, \mathtt{u}_i\}$ of U containing all the `Principal` allocation sites representing the equivalence classes of the `Principal` objects encapsulated in `s` in the analysis model, where U indicates the set of all the `Principal` allocation sites pointed to by the elements of S in the points-to graph associated with the PCG. Computing $\psi_S(\mathtt{s})$ requires identifying all the `Principal` instance keys in $\overline{N}$ pointed to by the `principals Set` field in `s`.

As discussed in Section 5.1, given a `Principal` object `u` encapsulated in an authenticated `Subject s`, two pieces of information about `u` are important when `s` needs to be authorized to access a restricted resource: the fully qualified class name of `u` and the name of `u`. These two pieces of information can be obtained statically.

The fully qualified class name of `u` is easy to retrieve in the analysis model since that information is stored into the allocation-site representation of `u` created by SARA. Let C indicate the set of all the fully qualified class names of the `Principal` allocation sites in U. Then function $\upsilon_C : U \to C$ maps each `Principal` allocation site to its fully qualified class name.

All the non-abstract classes inheriting from the `Principal` interface must implement instance method `getName`. At run time, for each `Principal` object `u` encapsulated in an authenticated `Subject s`, the Java authorization system invokes `getName` on `u` every time `doAs` or `doAsPrivileged` is called with `s` as the `Subject` parameter. This is to determine the `Permission`s granted to `u` by $\mathcal{D}$, which are added to the `ProtectionDomain`s of the methods following `doAs` or `doAsPrivileged` on the stack. As a result, for any $\mathtt{u} \in U$, N contains at least one node representing the invocation of `getName` on `u`. It is possible that N contains more than one such node if `u` is found in different `getName` receiver sets in the PCG. Let

```
grant  Principal c_1 "t_1"  ...  Principal c_h "t_h" {
          permission p_1;  ...  permission p_k;
};
```

Figure 2 Generic `Principal`-based `grant` Statement.

1: **for** $l := 1$ **to** h **do**
2:   **if** $\psi_{\mathtt{s},g}(\mathtt{c}_l) = \varnothing$ **then**
3:     **return** $\varnothing$
4: **for** $l := 1$ **to** h **do**
5:   **boolean** *found* := **false**
6:   **for each** $x \in \psi_{\mathtt{s},g}(\mathtt{c}_l)$ **do**
7:     **if** $\mathtt{t}_l \in \upsilon_T(\mathtt{u}_{rx})$ **then**
8:       *found* := **true**
9:       **break**
10:   **if** *found* = **false then**
11:     **return** $\varnothing$
12: **return** $\{p_1\} \sqcup \{p_2\} \sqcup \ldots \sqcup \{p_k\}$

Figure 3 Algorithm to Compute $\Sigma|_{S\times\mathcal{D}}(\mathtt{s}_r, g)$.

T be the set of all the `String` constants that have flowed to the return values of all the `Principal getName` PCG nodes. For any $\mathtt{u} \in U$, the set $\upsilon_T(\mathtt{u})$ of all the elements of T that could have flowed to the return values of all the `getName` PCG nodes having `u` in the receiver set is identified.

Let g be any `grant` statement in $\mathcal{D}$, similar to those shown in Figure 1. In general, g will be of a form similar to the one in Figure 2, which asserts that a `Subject s` is granted `Permissions` $\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_k \in P$ if `s` encapsulates at least h `Principal` objects with fully qualified class names $\mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_h \in C$ and names $\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_h \in T$, respectively, where the number of such `Principal` objects may be less than h if $\exists l_1, l_2 \in \{1, 2, \ldots, h\} : (\mathtt{c}_{l_1}, \mathtt{t}_{l_1}) = (\mathtt{c}_{l_2}, \mathtt{t}_{l_2})$.

Let $n \in D$ be the node examined above, with $\psi_D(n) = \{\mathtt{s}_1, \mathtt{s}_2, \ldots, \mathtt{s}_j\} \in \mathcal{P}(S)$. For $r \in \{1, 2, \ldots, j\}$, ψ_S can be applied to $\mathtt{s}_r$, and the result will be of the form $\psi_S(\mathtt{s}_r) = \{\mathtt{u}_{r1}, \mathtt{u}_{r2}, \ldots, \mathtt{u}_{ri_r}\} \in \mathcal{P}(U)$. This allows computing $\Sigma(n)$, the subset of $\overline{P_\equiv}$ representing the subject-granted access rights under which the `doAs` or `doAsPrivileged` method represented by n, in the context specified by n, will be executed at run time. First of all, to detect whether or not the access rights granted by g apply to n, it is sufficient to proceed as follows. Let $\psi_{\mathtt{s},g} : \{\mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_h\} \to \mathcal{P}(\{1, 2, \ldots, i_r\})$ be defined by:

$$\psi_{\mathtt{s},g}(\mathtt{c}_l) = \{x \in \{1, 2, \ldots, i_r\} : \upsilon_C(\mathtt{u}_{rx}) = c_l\}, \forall l \in \{1, 2, \ldots, h\}$$

Computing $\psi_{\mathtt{s},g}$ can be accomplished with one iteration over the elements of $\psi_S(\mathtt{s}_r)$. It should be observed also that $\psi_{\mathtt{s},g}$ partitions the set of indices $\{1, 2, \ldots, i_r\}$ in equivalence classes. Specifically, two indices $x, y \in \{1, 2, \ldots, i_r\}$ are in the same equivalence class if and only if either $\exists l \in \{1, 2, \ldots, h\} : \upsilon_C(\mathtt{u}_{rx}) = \upsilon_C(\mathtt{u}_{ry}) = \mathtt{c}_l$ or $\upsilon_C(\mathtt{u}_{rx}), \upsilon_C(\mathtt{u}_{ry}) \notin \{\mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_h\}$.

Furthermore, let $\Sigma|_{S\times\mathcal{D}} : S \times \mathcal{D} \to \mathcal{P}(\overline{P_\equiv})$ be the function mapping a pair $(\mathtt{s}, g) \in S \times \mathcal{D}$ to the subset of $\overline{P_\equiv}$ representing the access rights granted by g to the `Principals` encapsulated in `s`. For any given $r \in \{1, 2, \ldots, j\}$, $\Sigma|_{S\times\mathcal{D}}(\mathtt{s}_r, g)$ can be computed using the algorithm in Figure 3, where, in the notation introduced in Section 4.1.2, $p_1, p_2, \ldots, p_k$ represent the permission lattice elements corresponding to the `Permission` objects $\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_k$, respectively.

Lines 1–3 of the algorithm presented in Figure 3 simply verify that for each `Principal`

class name $\mathtt{c}_l$ specified in g, there is at least one `Principal` encapsulated in $\mathtt{s}_r$ whose class has that name. If this is not the case, then none of the `Permissions` listed in g can be applied to `s`, and $\Sigma|_{S\times\mathcal{D}}(\mathtt{s}_r, g) = \varnothing$.

Lines 4–11 are used to verify that for every `Principal` class name $\mathtt{c}_l$ specified in g, there exists at least one `Principal`, among those encapsulated in $\mathtt{s}_r$, with name $\mathtt{t}_l$ and class name $\mathtt{c}_l$. If this is not the case, then, once again, none of the `Permissions` listed in g can be applied to `s`, and $\Sigma|_{S\times\mathcal{D}}(\mathtt{s}_r, g) = \varnothing$. The presence of the **return** statements in Lines 3 and 11 guarantees that Line 12 is executed iff $\mathtt{s}_r$ encapsulates at least h `Principal` objects with fully qualified class names $\mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_h \in C$ and names $\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_h \in T$, respectively, in which case $\mathtt{s}_r$ is granted all the `Permissions` listed in g, as desired.

If multiple `grant` statements in $\mathcal{D}$ apply to $\mathtt{s}_r$, the permission-set lattice elements from all those `grant` statements must be joined, and the resulting permission-set lattice element, $\Sigma|_S(\mathtt{s}_r)$, represents the access rights granted to $\mathtt{s}_r$, as follows:

$$\Sigma|_S(\mathtt{s}_r) = \bigsqcup_{g\in\mathcal{D}} \Sigma|_{S\times\mathcal{D}}(\mathtt{s}_r, g)$$

This defines function $\Sigma|_S : S \to \mathcal{P}(\overline{P_\equiv})$. SARA cannot statically determine which `Subject` allocation site in the parameter set $\psi_D(n) = \{\mathtt{s}_1, \mathtt{s}_2, \ldots, \mathtt{s}_j\}$ will be passed as a parameter to the `doAs` or `doAsPrivileged` method represented by n at run time. Therefore, when computing the subject-granted access rights under which the method call represented by n is executed, the safest albeit most conservative assumption is to apply the meet operation to all the permission-set lattice elements associated with the `Subject` allocation sites $\mathtt{s}_1, \mathtt{s}_2, \ldots, \mathtt{s}_j$ as $\Sigma(n) = \sqcap_{r=1}^{j} \Sigma(\mathtt{s}_r)$.

5.3 Fixed-point Iteration

When `doAs` or `doAsPrivileged` is invoked with a `Subject` parameter `s`, the `Permissions` granted to the `Principals` associated with `s` are added to the stack frame corresponding to `doAs` or `doAsPrivileged`, and to those corresponding to all the downstream calls. A static model of this aspect requires propagating $\Sigma|_S(\mathtt{s})$ to all the descendants of the `doAs` or `doAsPrivileged` node in the PCG.

For any node $n \in N$, let $\mathrm{Gen_S}(n)$ and $\mathrm{Kill_S}(n)$ indicate the subsets of $\overline{P_\equiv}$ whose elements correspond to the subject-granted access rights *generated* and *killed*, respectively, by node n. If $n \in D$, then $\mathrm{Gen_S}(n) = \Sigma(n)$, otherwise $\mathrm{Gen_S}(n) = \varnothing$. To compute $\mathrm{Kill_S}(n)$, it should be noted that:

1. It is possible to call `doAs` multiple times within a thread, but only one `Subject` may be active at a time. If `doAs` is called with `Subject` parameter `s`, and later in the same thread there is another call to `doAs` with `Subject` parameter `t`, any subsequent authorization check will be performed with the authorizations granted to `t`, not those granted to `s`. The effect of the second call to `doAs` is to overwrite the authorizations granted to `s` until the second `doAs` call completes and returns to its caller. Similar considerations apply to `doAsPrivileged`.
2. In the absence of `Subjects` associated with a thread, a call to `doPrivileged` causes `checkPermission` to stop the stack walk at the stack frame corresponding to the method that called `doPrivileged`. The same principle applies when a `Subject` is present. If a `doPrivileged` call is made after invoking `doAs` or `doAsPrivileged` with `Subject` parameter `s`, `s` is not visible when an authorization check is performed by a `checkPermission` call. The effect of calling `doPrivileged` is to overwrite the authorizations granted to `s`.

Therefore, for any `doAs`, `doAsPrivileged`, or `doPrivileged` node $n \in N$, $\text{Kill}_S(n)$ is the universe $\overline{P_\equiv}$. For all other nodes $n \in N$, $\text{Kill}_S(n) = \varnothing$. This defines the two functions $\text{Gen}_A, \text{Kill}_A : N \to \mathcal{P}(\overline{P_\equiv})$.

SARA's *data-flow equation system*, for each node $n \in N$, is defined as follows:

$$\text{Out}_S(n) = \text{Gen}_S(n) \sqcup (\text{In}_S(n) - \text{Kill}_S(n)) \quad (1)$$

$$\text{In}_S(n) = \bigsqcap_{m \in \Gamma^-(n)} \text{Out}_S(m) \quad (2)$$

where $\text{Out}_S(n)$ and $\text{In}_S(n)$ are the subsets of $\overline{P_\equiv}$ corresponding to the subject-granted access rights *propagated* from and *reaching* n, respectively, and $\Gamma^- : N \to \mathcal{P}(N)$ is the function that maps each node in the PCG to the set of its predecessors. The recursive computation of functions $\text{In}_S, \text{Out}_S : N \to \mathcal{P}(\overline{P_\equiv})$ performed by resolving System (1,2) converges to a fixed point with a worst-case complexity of $\mathcal{O}(|E| \cdot \mathcal{H}(\mathcal{P}(\overline{P_\equiv})))$ [17]. In particular, convergence of System (1,2) is guaranteed by the fact that $\forall n \in N$, the *data-flow function* $\beta_n : \mathcal{P}(\overline{P_\equiv}) \to \mathcal{P}(\overline{P_\equiv})$ that transforms $\text{In}_S(n)$ into $\text{Out}_S(n)$ (i) is defined on a complete lattice, $(\mathcal{P}(\overline{P_\equiv}), \sqcap, \sqcup)$, with finite height; and (ii) is monotonic with respect to the lattice's partial order, $\sqsupseteq$. The worst-case complexity is $\mathcal{O}(|E| \cdot \mathcal{H}(\mathcal{P}(\overline{P_\equiv})))$ because each edge of the PCG will need to be traversed $\mathcal{H}(\mathcal{P}(\overline{P_\equiv}))$ times in the worst-case scenario. Given that $\mathcal{H}(\mathcal{P}(\overline{P_\equiv})) \in \mathcal{O}(|P|)$, the worst-case complexity can be expressed as $\mathcal{O}(|E||P|)$.

The fact that the forward propagation of permission-set lattice elements is initialized *only* with the nodes in D significantly reduces the time complexity in typical cases, since large portions of the PCG are likely not to contain any `doAs` or `doAsPrivileged` call and will never be affected by the fixed-point iteration generated by System (1,2).

5.4 PCG Annotation

When the data-flow algorithm just described terminates, each node $n \in N$ can be labelled with the subset $\Sigma(n)$ of $\overline{P_\equiv}$ defined by $\Sigma(n) = \text{Out}_S(n)$, which overapproximates the subject-granted access rights under which the method represented by n, in the calling context specified by n, will be executed at run time. Figure 4 shows a PCG annotated with the values of function Σ. The PCG contains three calls to `doAs`, one to `doPrivileged`, and three to `checkPermission`. The three calls to `doAs` take parameter pairs $(\texttt{s}, \texttt{a})$, $(\texttt{t}, \texttt{b})$, and $(\texttt{s}, \texttt{c})$, respectively, where `s` and `t` are `Subject` instances, and `a`, `b`, and `c` are `PrivilegedAction` instances. The three calls to `checkPermission` take arguments `p`, `q`, and `r` respectively, all of type `Permission`. The PCG annotation shows how the call to `doPrivileged` eliminates from the `AccessControlContext` the access rights granted to `s`.

It may be desirable to compute the subject-granted access rights under which a method can be executed assuming all the calling contexts that are realizable in the program under analysis. A function $\mu_S : M \to \mathcal{P}(\overline{P_\equiv})$ can be defined that maps each method `n` reachable from an entry point of the program to the subset of $\overline{P_\equiv}$ representing the subject-granted access rights under which `n` can be executed. Specifically, $\mu_S(\texttt{n})$ is defined as $\mu_S(\texttt{n}) = \sqcap_{i=1}^{k} \Sigma(n_i)$, where $\{n_1, n_2, \ldots, n_k\} = \nu(\texttt{n})$. The safest though most conservative approach is to compute the meet of the subsets $\Sigma(n_1), \Sigma(n_2), \ldots, \Sigma(n_k)$ of $\overline{P_\equiv}$, rather than their join. The reason is that, during the analysis, it is not known which calling context, among those of $n_1, n_2, \ldots, n_k$, `n` will be invoked with. As a result, it is not known which `Subject` will execute `n`.

5.5 Detection of PLP Violations

Unnecessary or redundant `doAs` and `doAsPrivileged` calls, as defined in Section 2.2, may lead to PLP violations and performance bottlenecks. As such, they should be removed.

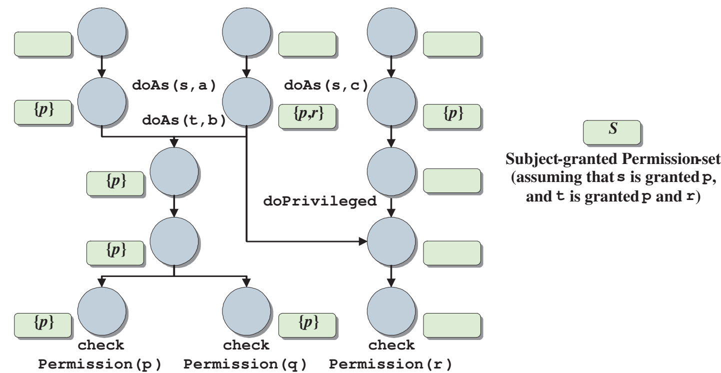


Figure 4 An Annotated PCG after SARA Completes.

SARA allows for detecting and flagging all such calls. To achieve this goal, SARA performs a basic access-rights analysis to detect the sets of permissions that each single method call will require at run time [9]. Any `doAs` or `doAsPrivileged` node that is mapped to ∅ represents a PLP violation.

6 Domain-specific Characteristics

This section describes the domain-specific characteristics implemented in the construction of a PCG in order to faithfully represent the subject-based permission model.

6.1 Modeling of Security-related Native Methods

Since SARA only analyzes Java bytecode, all *native* methods – those implemented in languages other than Java – would be incorrectly represented in the PCG as leaves, or nodes with no successors. However, many security analyses would not be sound without a model for those native methods that perform security-sensitive operations. Thus, the PCG is augmented with *automatically constructed* control- and data-flow-equivalent synthetic models [39] for a total of 162 native methods that are relevant to security analyses, such as:

- `Thread.start`, which executes a call to `Thread.run`
- The four forms of `AccessController.doPrivileged`
- `AccessController.getStackAccessControlContext`, which instantiates an object of type `AccessControlContext`, containing the `ProtectionDomains` of the classes whose methods are currently on the call stack

6.2 String Analysis

SARA is integrated with Path and Index-sensitive String Analysis (PISA) [34]. PISA can compute an over-approximation of the set of values that a variable of type `String` can take at run time. PISA is thus able to compute the set of `String` values that can flow to the target and action parameters of a `Permission` object. Without a string analysis capable of tracking `String` values and modeling operations on `String` objects, SARA would be unable to compute the target of the `Permission` object `p` in the following code snippet, and would conservatively have to overapproximate it as `"<<ALL_FILES>>"`, as explained in Section 3.2:

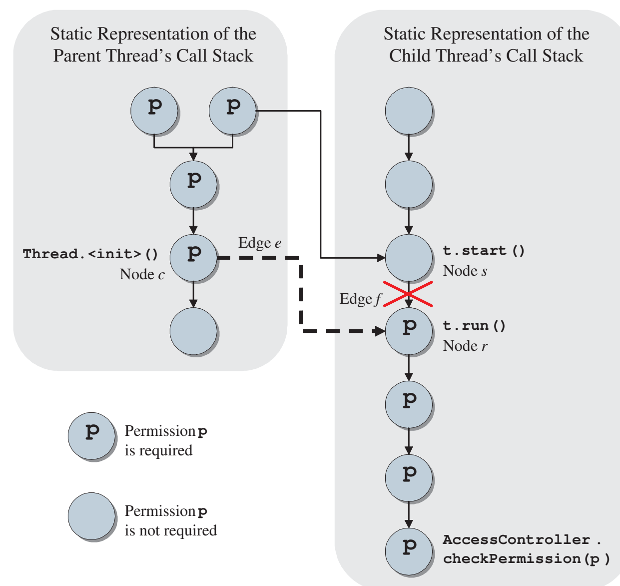


Figure 5 Domain-Specific PCG to Model Multi-threading.

```
String dir = "C:";
String file = "log.txt";
Permission p = new FilePermission(dir + File.separator + file, "read");
```

6.3 Modeling Multi-Threaded Code

In Java, to prevent confused-deputy attacks [20]. when access to a restricted resource is attempted from within a child thread, all the code in the child thread and all its ancestor threads must be granted the right to access that resource.

This behavior can be modelled by identifying all the `run()` nodes in the PCG $G = (N, E)$ whose receiver is a `Thread` object. For each such node r, with receiver `t`, the node c representing the invocation of the `Thread` constructor that instantiated `t` in the parent thread is identified, and a new edge $e = (c, r)$ is added to E. At the same time, edge $f = (s, r)$, where s represents the invocation of `start()` on `t`, is removed from E, as shown in Figure 5.

6.4 Extra Context for `Permission` Objects

The `Permission` parameter passed to `AccessController.checkPermission` is frequently instantiated by the `SecurityManager`. For example, when it is invoked by a library routine, the `SecurityManager`'s `checkWrite` method instantiates a `FilePermission` object and passes it to the `SecurityManager`'s `checkPermission` method, which finally passes it to the `checkPermission` method in the `AccessController` class. One problem is that different `FilePermission` objects instantiated through calls to `checkWrite` in different parts of the program will all share the same type and allocation site. Therefore, SARA would represent them as if they were the same object. As a result, different calls to `AccessController.checkPermission` would appear in the PCG as one node, yielding overly conservative results, as shown in Figure 6. Other `SecurityManager` access-control methods,

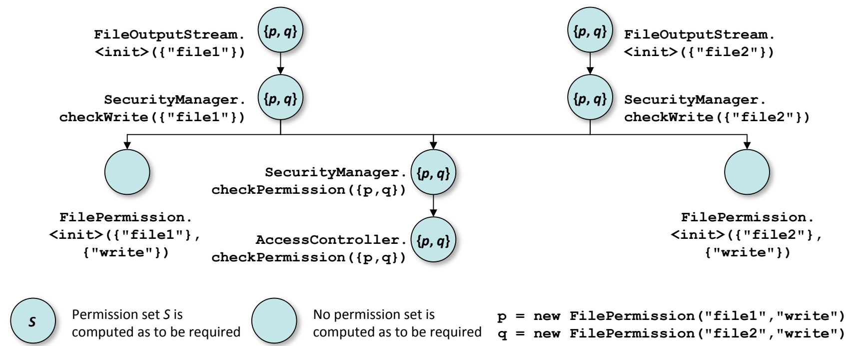


Figure 6 Conservativeness Introduced by the `SecurityManager`.

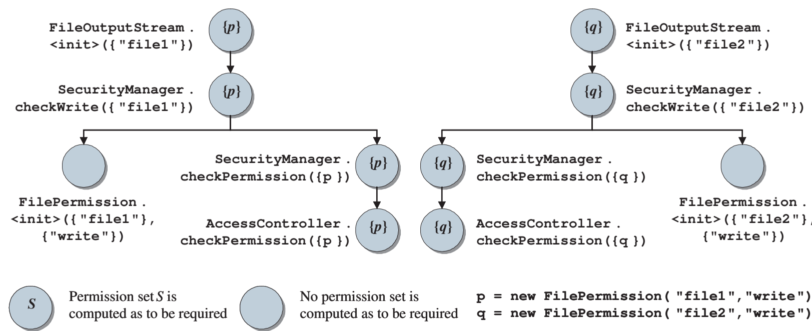


Figure 7 Extra Context for `Permission` Objects to Reduce Conservativeness.

such as `checkRead` and `checkConnect`, exhibit the same problem when represented in the PCG.

The solution is to distinguish `Permission` objects allocated in the `SecurityManager` access-control methods based not only on their types and allocation sites, but also on the nodes that contain those allocation sites. Therefore, if $m, n \in N$ are, for example, two `checkWrite` nodes in the PCG such that the parameters to the method calls they represent are the `Strings` `file1` and `file2`, respectively, the `FilePermission` allocated in m will be distinguished from the one allocated in n – even though both share the same type and allocation site – because $m \neq n$. Different `AccessController.checkPermission` calls resulting from `checkWrite` calls with different parameters will not appear in the PCG as one node. This allows a more precise propagation of permission-set lattice elements along different PCG paths, as shown in Figure 7.

To avoid building an unnecessarily large PCG, this specialization, which distinguishes `Permission` objects based on allocation sites with an additional level of calling-context sensitivity, is selectively applied only to `Permission` objects allocated in the `SecurityManager`.

6.5 Modeling `doAsPrivileged` Method Calls

In Java, the `AccessControlContext` passed to `doAsPrivileged` is obtained through a call to `getContext`. This method obtains the `AccessControlContext` associated with the current thread stack by calling `getStackAccessControlContext`. In order to faithfully propagate permission requirements along PCG paths containing a `doAsPrivileged` node d, it is sufficient to do the following:

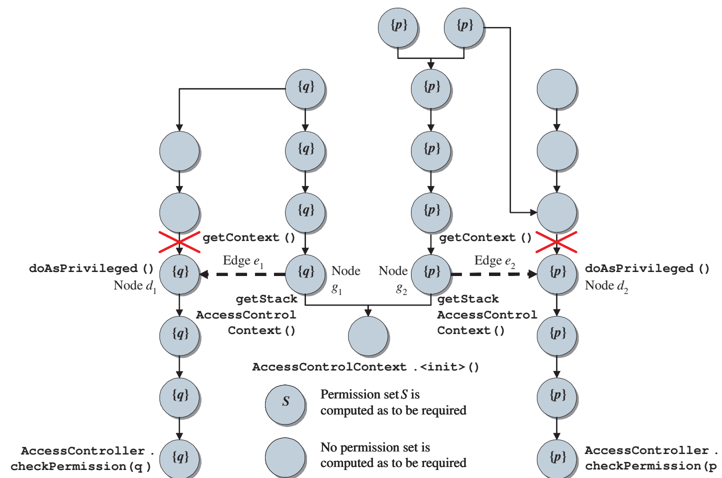


Figure 8 Modeling `doAsPrivileged` Method Calls.

1. Identify the `AccessControlContext` allocation sites $\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_k$ that may flow to the `AccessControlContext` parameter of the `doAsPrivileged` call represented by d
2. Locate the `getStackAccessControlContext` PCG nodes $g_1, g_2, \ldots, g_k \in N$ allocating $\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_k$, respectively
3. Remove d's preexisting predecessor edges from E
4. Add edges $e_1 = (g_1, d), e_2 = (g_2, d), \ldots, e_k = (g_k, d)$ to E

We observe that all the `AccessControlContext` instances obtained by calling `getContext` share the same allocation site, `a`, in method `getStackAccessControlContext`. Thus, in the analysis model, $\mathtt{a}_1 = \mathtt{a}_2 = \ldots = \mathtt{a}_k = \mathtt{a}$. Let d_1, d_2 be two PCG nodes representing calls to `doAsPrivileged`, and let $e_1 = (g, d_1), e_2 = (g, d_2)$ be two edges added to the PCG to model the propagation of permission requirements along the PCG paths. Since e_1, e_2 share the same tail node, g, when elements of $(\mathcal{P}(\overline{P_{\equiv}}), \sqcap, \sqcup)$ are propagated to g from d_1 and d_2, they are conservatively joined, yielding imprecise results. The solution consists of the following:

1. Adding one level of calling-context sensitivity to all the PCG nodes representing calls to `getStackAccessControlContext` and `getContext`
2. Distinguishing different `AccessControlContext` objects allocated through calls to method `getStackAccessControlContext` based not only on their allocation sites, but also on the nodes containing those allocation sites

Now, the PCG contains two `getStackAccessControlContext` nodes, g_1 and g_2, and two `getContext` nodes. Furthermore, SARA distinguishes between `AccessControlContext` objects $\mathtt{a}_1$ and $\mathtt{a}_2$, allocated in g_1 and g_2, respectively, as shown in Figure 8.

7 Evaluation

In this section, we describe an extensive experimental study.

7.1 Scope and Methodology

Subject-executed code is typically embedded in the business logic of enterprise application servers. This is because they service Web users, and at the same time they execute sensitive code and have access to critical organizational resources.

Therefore, our evaluation consisted of performing access-rights analysis of IBM WebSphere Application Server toward certifying it according to the CC EAL 4 standard. The application server contains over 2 MLOC, partitioned into 348 library components, which serve as our benchmarks. In order to pass the CC EAL 4 certification, it was necessary to correctly define access-control policies for each of these 348 libraries. For this to happen, each of the libraries was analyzed in isolation. As for the client code used for the analysis of each library, we utilized an existing testing harness in the form of a thorough unit-test suite, written by the application-server developers, which created a comprehensive client environment.

7.2 Experimental Results

Table 1 presents general information about the 348 benchmark libraries:

- 105 of the libraries were not originally associated with any policy at all (0 permissions granted to code and 0 to subjects).
- 141 had an associated access-control policy that only accounted for permissions granted to code (1,021), and ignored the permissions to be granted to subjects executing the code (0).
- The remaining 102 libraries had policies that accounted for both code (743) and subject permissions (132). For those 102 components, the subject-related permissions had been defined by developers based solely on manual code inspection and unit testing.

SARA achieved the following results:

Table 1 General Information about the Application Server Benchmarks.

Annotations	None	Code	Code & Subjects	Total
Library Count	105	141	102	348
`doAs` **Calls**	325	531	301	1,157
`doAsPriviledged` **Calls**	72	81	79	232
Original Code Permisssions	0	1,021	743	1,764
Original Subject Permissions	0	0	132	132
Permissions via SARA	143	191	151	485

- For the first 105 components, which required access-control policies for both the code and the subjects executing it, SARA computed 143 subject permissions.
- For the second set, of 141 components, SARA determined 191 subject permissions. Note that these permissions are subsequently subtracted from the sets of permissions granted to the code in order to prevent PLP violations.
- For the third and final category, SARA verified the policies that were constructed manually, and modified the permissions granted to the code in certain places in order to prevent permissions from being granted to both subject and code.

Table 1 also reports the number of `doAs` and `doAsPrivileged` calls. In total, there were 1,157 `doAs` and 232 `doAsPrivileged` calls, confirming that subject-executed code cannot be ignored when defining the access-control policy for an enterprise application server.

Table 2 highlights the specific violations detected by SARA on the 348 benchmarks:

- SARA detected 263 PLP violations, occurring when the same rights were granted to both code and subjects. This type of violation is only applicable to the third category of libraries – those that came with a developer-defined access-control policy accounting for both code and subjects. For the other two categories, no policy had been defined for the subjects executing the code, and so the number of PLP violations for those libaries was 0.

Table 2 Access-control Violations Detected by SARA.

Annotations	None	Code	Code & Subjects	Total
PLP Violations	0	0	263	263
Insufficient Policies	105	72	42	219
Unnecessary Calls	5	2	7	14
Redundant Calls	7	3	5	15

- SARA also detected 219 violations due to insufficient policies. This means that the policies coming with the library do not grant enough permissions to the subjects executing the code. It is interesting to note that the set of permissions that SARA computed for the subjects executing the code in the third library category was neither a superset nor a subset of the permissions originally computed by the developers. In some cases, permissions had been redundantly granted to both code and subjects, and so those permissions had to be removed. In other cases, the permissions were not listed at all, which would have caused authorization failures at run time (as confirmed during testing), and those permissions had to be added.

From a precision and performance point of view, we observe the following:

- The results produced by SARA were validated by dozens of developers, who confirmed them via a large set of dynamic test cases. None of the issues that were dynamically found was missed by SARA. False negatives may arise from not modeling native code and reflective calls, but SARA is equipped with (i) automatically generated analyzable synthetic artifacts for all the security-related methods in the application server and the underlying Java Runtime, and (ii) a mechanism for reflection resolution based on type cast information [26]. These two components significantly mitigate the risk of false negatives.
- SARA exhibited a high signal-to-noise ratio with 5% of the total number of reported violations being false positives.
- SARA was able to scale to, and analyze successfully, the entire application server, taking on average 103.45 seconds per library.

7.3 Discussion

Overall, SARA was able to detect a total of 511 access-rights flaws across the set of 348 benchmarks with an average scanning time of 103 seconds. We attribute SARA's ability to analyze each benchmark in under 2 minutes on average to its relatively inexpensive context-sensitivity policy. We could not compare SARA against other tools, unfortunately, as existing static-analysis algorithms can only identify the permissions required by a given component, but not those required by the subjects that will execute the code [23].

According to a thorough evaluation, which included a large set of dynamic test cases, SARA exhibited 0 false negatives. Having 0 false negatives was an important requirement for SARA because in access-rights analysis, a false negative corresponds to a missing permission, which at run time can cause unexpected authorization failures. Although we cannot claim that SARA is sound, our efforts to reduce false negatives by synthetically modeling native methods and disambiguating reflective calls through type cast information allow us to say that SARA is a *soundy* analysis [25].

SARA's false-positive rate was only 5%. This was established via careful scrutinization of the results, one by one, by the team developing the application server. A false positive

corresponds to an unnecessary permission, which constitutes a PLP violation, and so it was pertinent to identify all false positives before deploying the server.

PLP violations and vulnerabilities due to insufficient policies were corrected by simply modifying the policy files associated with the libraries. However, the only way to correct the remaining two vulnerabilities detected by SARA – unnecessary and redundant `doAs` and `doAsPrivileged` calls – was to change the source code of those libraries and remove such calls, which constitute both a PLP violation and a performance bottleneck. Our experience when communicating these vulnerabilities to the developers was that the interactions between the `doAs`, `doAsPrivileged`, `doPrivileged` and `checkPermission` APIs are very complicated and hard to explain to non-security experts, which is what caused these vulnerabilities to be present in the code in the first place. Having a tool for automatic detection of such unnecessary or redundant calls proved to be a crucial instrument to improve the quality of the code and teach developers how to use these APIs correctly in future code.

Based on these results, SARA proved to be accurate and useful. In our evaluation, it detected a large number of flaws, and scaled to all the libraries comprising the application server (2 MLOC in total), with an average analysis time of 103 seconds per library.

8 Related Work

There is no work on static analysis of subject-based authorization, particularly with regard to subject-granted rights analysis. Most of the work in the area of program analysis for access control has focused on computing permission requirements based on stack inspection, eliminating or minimizing redundant authorization tests, and defining alternatives to the current approach.

Pottier *et al.* [31] extend and formalize Wallach's security passing style [38] via type theory using a λ-calculus, called λ_{sec}. However, their work focuses only on basic authorization issues and is unable to perform incomplete-program analyses [32].

Jensen *et al.* [21] focus on proving that code is secure with respect to a global security policy. Their model uses operational semantics to prove the properties, via a two-level temporal logic, and shows how to detect redundant authorization tests. They assume all of the code is available for analysis, and that a call graph can be constructed for the code, though they do not do so themselves. Bartoletti *et al.* [6] are interested in optimizing performance of run-time authorization tests by eliminating redundant tests and relocating others as needed. Similarly, Banerjee and Naumann [5] apply denotational semantics to show the equivalence of "eager" and "lazy" semantics for stack inspection, provide a static analysis of *safety* (the absence of security errors), and identify transformations that can remove unnecessary authorization tests. These analyses are limited to a single thread and require the whole program.

Felten *et al.* have studied a number of security problems related to mobile code [36, 12, 38, 10, 37, 11]. In particular, they present a formalization of stack introspection. An authorization optimization technique, called *security passing style*, encodes the security state of an application while the application is executing [38]. The goal is to optimize authorization performance. Rather than analyzing security policies as embodied by existing code, Erlingsson and Schneider [13] describe a system that inlines reference monitors into the code to enforce specific security policies. The objective is to define a security policy and then inject authorization points into the code. This approach can reduce or eliminate redundant authorization tests. Conversely, this paper examines the authorization issue from the perspective of an existing system containing authorization test points.

Koved *et al.* [23] describe an algorithm and an actual implementation for correctly identifying the authorizations needed by a Java program, but do not deal with subject-executed code.

9 Conclusion and Future Work

In this paper, we have investigated the problem of performing comprehensive access-rights analysis. Such an analysis must account for subjects, which none of the existing permission analyses supports. We have validated the significance of this dimension experimentally via SARA, the first static permission analysis featuring subject sensitivity and policy correction/synthesis. The SARA algorithm features novel mechanisms to (i) represent the program, (ii) recover the permission hierarchy, and (iii) associate `Subjects` with `Permissions`. In the future we plan to enable SARA as an IDE tool. This requires real-time performance and incremental analysis capabilities, which we are currently developing.

References

1. M. Abadi and C. Fournet. Access Control Based on Execution History. In *NDSS*, 2003.
2. O. Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism. In *ECOOP*, 1995.
3. P. Anderson, T. Reps, and T. Teitelbaum. Design and Implementation of a Fine-Grained Software Inspection Tool. *TSE*, 29(8), 2003.
4. K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android Permission Specification. In *CCS*, 2012.
5. A. Banerjee and D. A. Naumann. Stack-based Access Control and Secure Information Flow. *JFP*, 15(2), 2005.
6. M. Bartoletti, P. Degano, and G. L. Ferrari. Static Analysis for Stack Inspection. In *ConCoord*, volume 54, 2001.
7. F. Besson, T. Blanc, C. Fournet, and A. D. Gordon. From Stack Inspection to Access Control: A Security Analysis for Libraries. In *CSFW*, 2004.
8. P. Centonze. *An Algebra for Access Control*. PhD thesis, New York University, Polytechnic School of Engineering, Brooklyn, NY, USA, 2008.
9. P. Centonze, R. J. Flynn, and M. Pistoia. Combining Static and Dynamic Analysis for Automatic Identification of Precise Access-control Policies. In *ACSAC*, 2007.
10. D. Dean. The Security of Static Typing with Dynamic Linking. In *Proceedings of the 4th ACM Conference on Computer and Communications Security (CCS)*, 1997.
11. D. Dean, E. W. Felten, and D. S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *S&P*, 1996.
12. R. D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, Princeton, NJ, USA, 1999.
13. Ú. Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *S&P*, 2000.
14. A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *CCS*, 2011.
15. C. Fournet and A. D. Gordon. Stack Inspection: Theory and Variants. In *POPL*, 2002.
16. E. Geay, M. Pistoia, T. Tateishi, B. G. Ryder, and J. Dolby. Modular String-sensitive Permission Analysis with Demand-driven Precision. In *ICSE*, 2009.
17. G. Grätzer. *General Lattice Theory*. Birkhäuser, second edition, 2003.
18. D. Grove and C. Chambers. A Framework for Call Graph Construction Algorithms. *TOPLAS*, 23(6), 2001.

19 S. Gulwani and G. C. Necula. Path-sensitive Analysis for Linear Arithmetic and Uninterpreted Functions. In *SAS*, 2004.
20 N. Hardy. The Confused Deputy (or Why Capabilities Might Have Been Invented). *OSR*, 22(4), 1988.
21 T. P. Jensen, D. Le Métayer, and T. Thorn. Verification of Control Flow Based Security Properties. In *S&P*, 1999.
22 G. A. Kildall. A Unified Approach to Global Program Optimization. In *POPL*, 1973.
23 L. Koved, M. Pistoia, and A. Kershenbaum. Access Rights Analysis for Java. In *OOPSLA*, 2002.
24 C. Lai, L. Gong, L. Koved, A. J. Nadalin, and R. Schemers. User Authentication and Authorization in the Java™ Platform. In *ACSAC*, 1999.
25 B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In Defense of Soundiness: A Manifesto. *CACM*, 58(2), 2015.
26 B. Livshits, J. Whaley, and M. S. Lam. Reflection Analysis for Java. In *APLAS*, 2005.
27 S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.
28 G. Naumovich. A Conservative Algorithm for Computing the Flow of Permissions in Java Programs. In *ISSTA*, 2002.
29 M. Pistoia. *A Unified Mathematical Model for Stack- and Role-Based Authorization Systems*. PhD thesis, New York University, Polytechnic School of Engineering, Brooklyn, NY, USA, 2005.
30 M. Pistoia, A. Banerjee, and D. A. Naumann. Beyond Stack Inspection: A Unified Access-control and Information-flow Security Model. In *S&P*, 2007.
31 F. Pottier, C. Skalka, and S. F. Smith. A Systematic Approach to Static Access Control. In *ESOP*, 2001.
32 B. G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Languages. In *CC*, 2003. Invited Paper.
33 J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, volume 63, 1975.
34 T. Tateishi, M. Pistoia, and O. Tripp. Path- and Index-sensitive String Analysis Based on Monadic Second-order Logic. *TOSEM*, 22(4), 2013.
35 O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *FASE*, 2013.
36 D. S. Wallach. *A New Approach to Mobile-Code Security*. PhD thesis, Princeton University, Princeton, NJ, USA, 1999.
37 D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architectures for Java. In *SOSP*, 1997.
38 D. S. Wallach and E. W. Felten. Understanding Java Stack Inspection. In *S&P*, 1998.
39 X. Zhang, L. Koved, M. Pistoia, S. Weber, T. Jaeger, G. Marceau, and L. Zeng. The Case for Analysis Preserving Language Transformation. In *ISSTA*, 2006.

Variability Abstractions: Trading Precision for Speed in Family-Based Analyses*

Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski

IT University of Copenhagen
Denmark
`{adim,brabrand,wasowski}@itu.dk`

Abstract

Family-based (lifted) data-flow analysis for Software Product Lines (SPLs) is capable of analyzing all valid products (variants) without generating any of them explicitly. It takes as input only the common code base, which encodes all variants of a SPL, and produces analysis results corresponding to all variants. However, the computational cost of the lifted analysis still depends inherently on the number of variants (which is exponential in the number of features, in the worst case). For a large number of features, the lifted analysis may be too costly or even infeasible. In this paper, we introduce variability abstractions defined as Galois connections and use abstract interpretation as a formal method for the calculational-based derivation of approximate (abstracted) lifted analyses of SPL programs, which are sound by construction. Moreover, given an abstraction we define a syntactic transformation that translates any SPL program into an abstracted version of it, such that the analysis of the abstracted SPL coincides with the corresponding abstracted analysis of the original SPL. We implement the transformation in a tool, that works on Object-Oriented Java program families, and evaluate the practicality of this approach on three Java SPL benchmarks.

1998 ACM Subject Classification F.3.2 [Logics and Meanings of Programs] Semantics of Programming Languages – Program analysis

Keywords and phrases Software Product Lines, Family-Based Program Analysis, Abstract Interpretation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.247

1 Introduction and Motivation

Software Product Lines (SPLs) are an effective strategy for developing and maintaining a family of related programs. Any valid program (*variant*) of an SPL is specified in terms of features selected. A *feature* is a distinctive aspect, quality, or characteristic from the problem-domain of a system. SPLs have been adopted by the industry because of improvements in productivity and time-to-market [7]. While there are many implementation strategies, many industrial product lines are implemented using annotative approaches such as conditional compilation; in particular, via the C-preprocessor `#ifdef` construct [17].

Recently, formal analysis and verification of SPLs have been a topic of considerable research (see [23] for a survey). The challenge is to develop analysis and verification techniques that work at the level of program families, rather than the level of individual programs. Given that the number of variants grows exponentially with the number of features, the need for efficient

* Partially supported by The Danish Council for Independent Research under a Sapere Aude project, VARIETE.

© Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 247–270

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

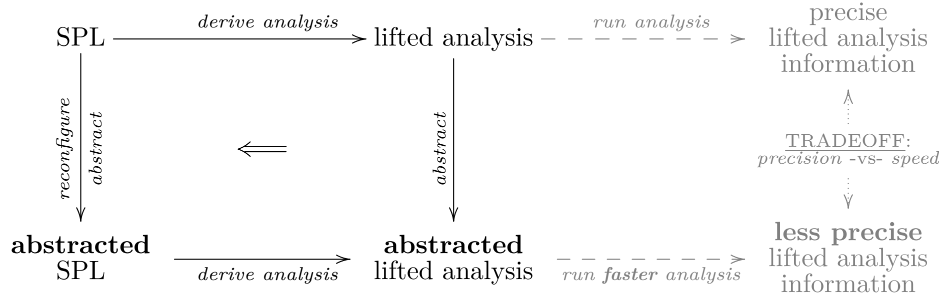


Figure 1 Diagram illustrating the role and intended usage of the `reconfigurator` transformation. Instead of abstracting an already existing (or derived) lifted analysis, our transformation allows abstraction to be applied directly to the SPL. The resulting "abstracted SPL" can then be analyzed using existing techniques. The two paths from SPL to "abstracted lifted analysis" are guaranteed to produce the same abstracted lifted analysis.

analysis and verification techniques is essential. To address this, a number of so-called *lifted* techniques have emerged, essentially lifting existing analysis and verification techniques to work on program families, rather than on individual programs. This includes lifted type checking [18], lifted data-flow analysis [5, 4], lifted model checking [6]. They are also known as family-based (*variability-aware* or feature-sensitive) techniques. Lifted techniques are capable of analyzing the entire code base (all variants at once), without having to explicitly generate and analyze all individual variants, one at a time. Also, lifted techniques are capable of pin-pointing errors directly in the product line, as opposed to reporting errors in an individual product derived from the SPL.

There are two ways to speed up analyses: improving *representation* and increasing *abstraction*. The former has received considerable attention in the field of family-based analysis. In this paper, we investigate the latter. We consider a range of abstractions at the *variability level* that may tame the combinatorial explosion of configurations and reduce it to something more tractable by manipulating the configuration space of a program. Such variability abstractions enable deliberate trading of precision for speed in family-based analyses, even turn infeasible analyses into feasible ones, while retaining an intimate relationship back to the original analysis (via the abstraction).

We organize our variability abstractions in a calculus that provides convenient, modular, and compositional declarative specification of abstractions. We propose two basic abstraction operators (*project* and *join*) and two compositional abstraction operators (*sequential composition* and *parallel composition*). Each abstraction expresses a compromise between precision and speed in the induced abstracted analysis. We show how to apply each of these abstractions to data-flow lifted analyses, to derive their corresponding efficient and sound (correct) abstracted lifted analysis based on the calculational approach of abstract interpretation developed in [9]. Moreover, we present a method for extracting data-flow equations corresponding to each abstracted analysis. Note that the approach is applicable to *any* analysis phrased as an abstract interpretation; in particular, it is not limited to data-flow analysis.

We observe that for variability abstractions, *analysis abstraction* and *analysis derivation* commute. Figure 1 illustrates how analysis abstraction is classically undertaken and how we propose to optimize it. The top left corner shows a product line that we want to analyze. A lifted analyzer will take an SPL as input and derive a "lifted analysis" (rightward

arrow). We can then run that lifted analysis (next rightward dashed arrow) and obtain our "*precise* lifted analysis information". (Note that for some analyzers, the phases *derive analysis* and subsequent *run analysis* may be so intertwined that they are not independently distinguishable.) Since running the analysis might be too slow or infeasible, we may decide to use abstraction to obtain a faster, although less precise analysis. Classically, an abstraction is applied to the derived analysis before it is run (middle arrow down) which, after an often long and complex process, produces an "abstracted lifted analysis". When that analysis is subsequently run, it will produce less precise analysis information, but it will do so faster than the original analysis (i.e., there is a *precision vs. speed tradeoff*).

Interestingly, for lifted analyses and variability abstractions, the analysis abstraction (down) and derivation (right) commute and we may swap their order of application, as indicated by the short double leftward arrow in the center. The implications are quite significant. It means that variability abstractions can be applied *before*, and independently of, the subsequent analysis. This also means that the same variability abstractions might be applicable to all sorts of analyses that are specifiable via abstract interpretation; including, but not limited to: data-flow analysis [10], model checking [12], type systems [8] and testing [15].

We exploit this observation to define a *stand-alone* source-to-source transformation, called `reconfigurator`, for programs with `#ifdef`s. It takes an input SPL program and a variability abstraction and produces an abstracted SPL program such for which the subsequent lifted analysis agrees with "abstracted lifted analysis" of the original unabstracted SPL. Like a preprocessor the `reconfigurator` is essentially unaware of the programming language syntax, thus it can be used for any analysis. Many existing analysis methods that are unable to abstract variability benefit from this work instantly. Almost no extension or adaptation is required as the abstraction is applied to source code before analysis.

We evaluate our approach by comparing analyses of a range of increasingly abstracted SPLs against their origins without abstraction, quantifying to what extent precision can be traded for speed in lifted analyses.

In summary, the paper makes the following contributions:

- **C1:** *Variability abstraction* as a method for trading precision for speed in family-based analysis (based on abstract interpretation);
- **C2:** A *calculus* for modular specification of variability abstractions;
- **C3:** The observation that certain analysis derivations and analysis abstractions *commute*, meaning that variability abstractions can be applied directly on an SPL *before* (and independently of) subsequent lifted analysis;
- **C4:** A stand-alone *transformation*, `reconfigurator`, based on the above ideas;
- **C5:** An *evaluation* of the above ideas; in particular, an evaluation of the tradeoff between precision and speed in family-based analyses.

We direct this work to program analysis and software engineering researchers. The method of *variability abstractions* (**C1**–**C3**) is directed at designers of lifted analyses for product lines. They may use our insights to design improved abstracted analyses that appropriately trade precision for speed. Note that the ideas apply beyond the context of data-flow analyses (e.g., to model checking, type systems, verification, and testing). The `reconfigurator` (**C4**) and the evaluation lessons (**C5**) are relevant for software engineers working on preprocessor-based product lines and who would like to speed up existing analyzers.

We proceed by introducing the basics of lifting analyses in Section 2. Section 3 defines a calculus for specification of variability abstractions. Section 4 explains how to apply an abstraction to a lifted analysis. It uses constant propagation as an example. The `reconfigurator` is described in Section 5 along with correctness for our example analysis.

Section 6 presents the evaluation on three Java Object-Oriented SPLs. Finally, we discuss the relation to other works and conclude.

2 Program Families and Lifted Analyses

In this section we summarize the prerequisites for presenting our work. We define *features*, *configurations*, *feature expressions*, and a *feature model* which designates a set of *valid* configurations. Hereafter, we describe a simple imperative language $\overline{\text{IMP}}$ for writing program families. Finally, we briefly sketch a lifted constant propagation analysis for this language, formally derived in [19]. We focus on constant propagation for presentation purposes; our approach is generically applicable to any lifted analysis phrased as an abstract interpretation.

Features, Configurations, and Feature Expressions. Let $\mathbb{F} = \{A_1, \ldots, A_n\}$ be a finite set of *features*, each of which may be *enabled* or *disabled* in a particular program variant. A *feature expression*, *FeatExp* formula, is a propositional logic formula over $\mathbb{F}$, defined inductively by:

$$\varphi \quad ::= \quad A \in \mathbb{F} \quad | \quad \neg\varphi \quad | \quad \varphi_1 \wedge \varphi_2$$

A truth assignment or *valuation* is a mapping v assigning a truth value to all features. Every feature expression evaluates to some truth value under the valuation v. We say that φ is *valid*, denoted as $\models \varphi$, if φ evaluates to *true* for all valuations v. We say that φ is *satisfiable*, denoted as $\text{sat}(\varphi)$, if there exists a valuation v such that φ evaluates to true under v. We say that the formula θ is a semantic consequence of φ, denoted as $\varphi \models \theta$, if for all satisfiable valuations v of φ it follows that θ evaluates to *true* under v. Otherwise, we have $\varphi \not\models \theta$.

Feature Model. A feature model describes the set of *valid* configurations (variants) of a product line in terms of features and relationships among them. For our purposes a feature model can be equated to a propositional formula [2], say $\psi \in \mathit{FeatExp}$, as the semantic aspects of feature models beyond the configuration semantics, are not relevant here. We write $\mathbb{K}_\psi$ to denote the set of all *valid* configurations described by the feature model, ψ; i.e., the set of all satisfiable valuations of ψ. One satisfiable valuation v represents a valid configuration, and it can be also encoded as a conjunction of literals: $k_v = v(A_1) \cdot A_1 \wedge \cdots \wedge v(A_n) \cdot A_n$, where $\mathit{true} \cdot A = A$ and $\mathit{false} \cdot A = \neg A$, such that $k_v \models \psi$. The truth value of a feature in v indicates whether the given feature is *enabled* (included) or *disabled* (excluded) in the corresponding configuration. Let $k_{v_1}, \ldots, k_{v_n}$ $(1 \leq n \leq 2^{|\mathbb{F}|})$ represent all satisfiable valuations of ψ expressed as formulas, then $\mathbb{K}_\psi = \{k_{v_1}, \ldots, k_{v_n}\}$. For example, the set of features, $\mathbb{F} = \{A, B\}$, and the feature model, $\psi = A \vee B$, yield the following set of *valid* configurations: $\mathbb{K}_\psi = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$.

The Programming Language. $\overline{\text{IMP}}$ is an extension of the imperative language IMP [25] often used in semantic studies. $\overline{\text{IMP}}$ adds a compile-time conditional statement for encoding multiple variants of a program. The new statement "`#if` (θ) s" contains a feature expression $\theta \in \mathit{FeatExp}$ as a condition and a statement s that will be run, i.e. included in a variant, iff the condition θ is satisfied by the corresponding configuration $k \in \mathbb{K}_\psi$. The abstract syntax of the language is given by the following grammar:

$$s \quad ::= \quad \texttt{skip} \mid \texttt{x := } e \mid s \;\texttt{;}\; s \mid \texttt{if } e \texttt{ then } s \texttt{ else } s \mid \texttt{while } e \texttt{ do } s \mid \texttt{\#if } (\theta)\; s$$

$$e \quad ::= \quad n \mid \texttt{x} \mid e \oplus e$$

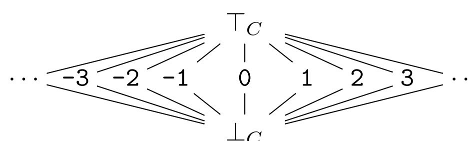
where n ranges over integers, $\mathtt{x}$ ranges over variable names *Var*, and $\oplus$ over binary arithmetic operators. The set of all generated statements s (respectively expressions e) is denoted by *Stm* (respectively *Exp*). Notice that $\overline{\text{IMP}}$ is only used for presentational purposes as a well established minimal language. Still, the introduced methodology is not limited to $\overline{\text{IMP}}$ or its features. In fact, we evaluate our approach on Object-Oriented program families written in Java.

The semantics of $\overline{\text{IMP}}$ has two stages. First, a preprocessor takes as input an $\overline{\text{IMP}}$ program and a configuration $k \in \mathbb{K}_\psi$, and outputs a variant, i.e. an IMP program without `#if`-s, corresponding to k. All "`#if` (θ) s" statements are appropriately resolved in the generated valid product, i.e. s is included in it iff $k \models \theta$. Then, the obtained variant is executed (compiled) using the standard IMP semantics [25].

Analysis Framework. In the context of $\overline{\text{IMP}}$ lifting means taking a static analysis that works on IMP programs, and transforming it into an analysis that works on $\overline{\text{IMP}}$ programs, without preprocessing them (so on all the variants simultaneously).

Suppose that we have a monotone data-flow analysis framework for single programs, and we want to lift the analysis to the family level setting. Let $\langle \mathbb{X}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ be a complete lattice with finite height, which represents a property domain suitable for performing a computable analysis for single programs. By using variational abstract interpretation [19], we can derive the corresponding lifted analysis for $\overline{\text{IMP}}$, whose lifted property domain is $\langle \mathbb{X}^{\mathbb{K}_\psi}, \dot\sqsubseteq, \dot\sqcup, \dot\sqcap, \dot\bot, \dot\top \rangle$ where $\mathbb{K}_\psi$ is the set of valid configurations. In the following, we use constant propagation analysis as an example to demonstrate our approach for deriving computationally cheap abstracted lifted analysis. Still, the approach is by no means limited to constant propagation, but it is applicable to any static analysis from the monotone data-flow analysis framework [20] that can be lifted.

Constant Propagation Analysis. We first define a constant propagation lattice $\langle Const, \sqsubseteq_C \rangle$, whose partial ordering $\sqsubseteq_C$ is given by:



In this domain $\top_C$ indicates a value which may be a *non-constant*, and $\bot_C$ indicates *unanalyzed* information. All other elements indicate constant values. The partial ordering $\sqsubseteq_C$ induces a least upper bound, $\sqcup_C$, and a greatest lower bound operator, $\sqcap_C$, on the lattice elements. For example, we have $0 \sqcup_C 1 = \top_C$, $\top_C \sqcap_C 1 = 1$, etc.

The constant propagation analysis is given in terms of abstract *constant propagation stores*, denoted by a, essentially mappings of variables to elements of *Const*. Thus $a(\mathtt{x})$ informs whether the variable $\mathtt{x}$ is a constant, and, in this case, what is its value. We write $\mathbb{A} = Var \to Const$ meaning the domain of all constant propagation stores. Since *Const* is a complete lattice then so is $\langle \mathbb{A}, \sqsubseteq_\mathbb{A}, \sqcup_\mathbb{A}, \sqcap_\mathbb{A}, \bot_\mathbb{A}, \top_\mathbb{A} \rangle$ obtained by point-wise lifting [25]. For example, for $a, a' \in \mathbb{A}$ we have $a \sqsubseteq_\mathbb{A} a'$ iff $\forall \mathtt{x} \in Var$, $a(\mathtt{x}) \sqsubseteq_C a'(\mathtt{x})$. We omit the subscripts C and $\mathbb{A}$ whenever they are clear in context.

Lifted Constant Propagation Analysis. For the lifted constant propagation analysis, we work with the lifted property domain $\langle \mathbb{A}^{\mathbb{K}_\psi}, \dot\sqsubseteq, \dot\sqcup, \dot\sqcap, \dot\bot, \dot\top \rangle$, where $\mathbb{A}^{\mathbb{K}_\psi}$ is shorthand for the

$$
\begin{aligned}
\overline{\mathcal{A}}[\![\texttt{skip}]\!] &= \lambda\overline{a}.\,\overline{a} \\
\overline{\mathcal{A}}[\![\texttt{x := } e]\!] &= \lambda\overline{a}.\prod_{k\in\mathbb{K}_\psi}(\pi_k(\overline{a}))[\texttt{x}\mapsto\pi_k(\overline{\mathcal{A}'}[\![e]\!]\overline{a})] \\
\overline{\mathcal{A}}[\![s_0\ \texttt{;}\ s_1]\!] &= \overline{\mathcal{A}}[\![s_1]\!]\circ\overline{\mathcal{A}}[\![s_0]\!] \\
\overline{\mathcal{A}}[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!] &= \lambda\overline{a}.\,\overline{\mathcal{A}}[\![s_0]\!]\overline{a}\,\dot\sqcup\,\overline{\mathcal{A}}[\![s_1]\!]\overline{a} \\
\overline{\mathcal{A}}[\![\texttt{while } e \texttt{ do } s]\!] &= \mathrm{lfp}\lambda\overline{\Phi}.\,\lambda\overline{a}.\,\overline{a}\,\dot\sqcup\,\overline{\Phi}(\overline{\mathcal{A}}[\![s]\!]\,\overline{a}) \\
\overline{\mathcal{A}}[\![\texttt{\#if }(\theta)\ s]\!] &= \lambda\overline{a}.\prod_{k\in\mathbb{K}_\psi}\begin{cases}\pi_k(\overline{\mathcal{A}}[\![s]\!]\overline{a}) & \text{if } k\models\theta \\ \pi_k(\overline{a}) & \text{if } k\not\models\theta\end{cases} \\
\overline{\mathcal{A}'}[\![n]\!] &= \lambda\overline{a}.\prod_{k\in\mathbb{K}_\psi}\texttt{n} \\
\overline{\mathcal{A}'}[\![\texttt{x}]\!] &= \lambda\overline{a}.\prod_{k\in\mathbb{K}_\psi}\pi_k(\overline{a})(\texttt{x}) \\
\overline{\mathcal{A}'}[\![e_0\oplus e_1]\!] &= \lambda\overline{a}.\prod_{k\in\mathbb{K}_\psi}\pi_k(\overline{\mathcal{A}'}[\![e_0]\!]\overline{a})\ \widehat{\oplus}\ \pi_k(\overline{\mathcal{A}'}[\![e_1]\!]\overline{a})
\end{aligned}
$$

Figure 2 Definitions of $\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A}\to\mathbb{A})^{\mathbb{K}_\psi}$ and $\overline{\mathcal{A}'}[\![e]\!] : (\mathbb{A}\to \mathit{Const})^{\mathbb{K}_\psi}$.

$|\mathbb{K}_\psi|$-fold product $\prod_{k\in\mathbb{K}_\psi}\mathbb{A}$, i.e. there is one separate copy of $\mathbb{A}$ for each valid configuration of $\mathbb{K}_\psi$. The ordering $\dot\sqsubseteq$ is lifted configuration-wise; i.e., for $\overline{a},\overline{a}'\in\mathbb{A}^{\mathbb{K}_\Psi}$ we have $\overline{a}\ \dot\sqsubseteq\ \overline{a}' \equiv_{def} \pi_k(\overline{a}) \sqsubseteq_{\mathbb{A}} \pi_k(\overline{a}')$ for all $k\in\mathbb{K}_\psi$. Here π_k selects the kth component of a tuple. Similarly, we lift configuration-wise all other elements of the complete lattice $\mathbb{A}$, obtaining $\dot\sqcup, \dot\sqcap, \dot\bot, \dot\top$. E.g., $\dot\top = \prod_{k\in\mathbb{K}_\psi}\top_{\mathbb{A}} = (\top_{\mathbb{A}},\ldots,\top_{\mathbb{A}})$.

The lifted analysis $\overline{\mathcal{A}}[\![s]\!]$ should be a function from $\mathbb{A}^{\mathbb{K}_\psi}$ to $\mathbb{A}^{\mathbb{K}_\psi}$. However, using a tuple of $|\mathbb{K}_\psi|$ independent simple functions of type $\mathbb{A}\to\mathbb{A}$ is sufficient. Thus, the lifted analysis is given by the function $\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A}\to\mathbb{A})^{\mathbb{K}_\psi}$, which represents a tuple of $|\mathbb{K}_\psi|$ functions of type $\mathbb{A}\to\mathbb{A}$. The k-th component of $\overline{\mathcal{A}}[\![s]\!]$ defines the analysis corresponding to the valid configuration described by the formula k. Thus, an analysis $\overline{\mathcal{A}}[\![s]\!]$ transforms a lifted store, $\overline{a}\in\mathbb{A}^{\mathbb{K}_\psi}$, into another lifted store of the same type. For simplicity, we overload the λ-abstraction notation, so creating a tuple of functions looks like a function on tuples: we write $\lambda\overline{a}.\prod_{k\in\mathbb{K}} f_k(\pi_k(\overline{a}))$ to mean $\prod_{k\in\mathbb{K}}\lambda a_k.f_k(a_k)$. Similarly, if $\overline{f} : (\mathbb{A}\to\mathbb{A})^{\mathbb{K}}$ and $\overline{a}\in\mathbb{A}^{\mathbb{K}}$, then we write $\overline{f}(\overline{a})$ to mean $\prod_{k\in\mathbb{K}}\pi_k(\overline{f})(\pi_k(\overline{a}))$.

The equations for lifted analysis $\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A}\to\mathbb{A})^{\mathbb{K}_\psi}$ and $\overline{\mathcal{A}'}[\![e]\!] : (\mathbb{A}\to \mathit{Const})^{\mathbb{K}_\psi}$ that analyse all valid configurations simultaneously are given in Fig 2. They are systematically derived in [19] by following the steps of the calculational approach to abstract interpretation [9]: define collecting semantics, specify a series of Galois connections and compose them with the collecting semantics to obtain the resulting analysis, which is thus sound (correct) by construction. Monotonicity of $\overline{\mathcal{A}}[\![s]\!]$ and $\overline{\mathcal{A}'}[\![e]\!]$ was shown in [19] as well.

The (transfer) function $\overline{\mathcal{A}}[\![s]\!]$ captures the effect of analysing the statement s in an input store $\overline{a}$ by computing an output store $\overline{a}'$. For the `skip` statement, the analysis function is an identity on lifted stores. For the assignment statement, `x :=` e, the value of variable `x` is updated in every component of the input store $\overline{a}$ by the value of the expression e evaluated in the corresponding component of $\overline{a}$. The `if` case results in the least upper bound (join) of the effects from the two corresponding branches, and it abstracts away the analysis information at the guard (condition) point. For the `while` statement, we compute the least fixed point

of a functional[1] in order to capture the effect of running all possible iterations of the `while` loop. This fixed point exists and is computable by Kleene's fixed point theorem, since the functional is a monotone function over complete lattice with finite height [19, 10]. For the `#if` (θ) s statement, we check for each valid configuration k [2] whether the feature constraint θ is satisfied and, if so, it updates the corresponding component of the input store by the effect of evaluating the statement s. Otherwise, the corresponding component of the store is not updated. The function $\overline{\mathcal{A}'}[\![e]\!]$ describes the result of evaluating the expression e in a lifted store. Note that, for each binary operator $\oplus$, we define the corresponding constant propagation operator $\widehat{\oplus}$, which operates on values from *Const*, as follows:

$$v_0 \mathbin{\widehat{\oplus}} v_1 = \begin{cases} \bot & \text{if } v_0 = \bot \vee v_1 = \bot \\ \mathtt{n} & \text{if } v_0 = \mathtt{n}_0 \wedge v_1 = \mathtt{n}_1,\ \text{where } \mathtt{n} = \mathtt{n}_0 \oplus \mathtt{n}_1 \\ \top & \text{otherwise} \end{cases}$$

We lift the above operation configuration-wise, and in this way obtain a new operation $\dot{\widehat{\oplus}}$ on tuples of *Const* values.

▶ **Example 1.** Consider the $\overline{\text{IMP}}$ program S_1:

$$\begin{aligned} &\mathtt{x} := 0; \\ &\mathtt{\#if}\ (A)\ \mathtt{x} := \mathtt{x} + 1; \\ &\mathtt{\#if}\ (B)\ \mathtt{x} := 1 \end{aligned}$$

with the set $\mathbb{K}_\psi = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$. By using the rules of Fig. 2, we can calculate $\overline{\mathcal{A}}[\![S_1]\!]$ for a store in which $\mathtt{x}$ is uninitialized, i.e. it has the value $\top$. We assume a convention here that the first component of the store corresponds to configuration $A \wedge B$, the second to $A \wedge \neg B$, and the third to $\neg A \wedge B$. We write $\overline{a_0} \stackrel{\overline{\mathcal{A}}[\![s]\!]}{\longmapsto} \overline{a_1}$ when $\overline{\mathcal{A}}[\![s]\!]\overline{a_0} = \overline{a_1}$. We have:

$$\begin{aligned} &\big([\mathtt{x}\mapsto\top],[\mathtt{x}\mapsto\top],[\mathtt{x}\mapsto\top]\big) \stackrel{\overline{\mathcal{A}}[\![\mathtt{x}:=0]\!]}{\longmapsto} \big([\mathtt{x}\mapsto 0],[\mathtt{x}\mapsto 0],[\mathtt{x}\mapsto 0]\big) \\ &\stackrel{\overline{\mathcal{A}}[\![\mathtt{\#if}\,(A)\,\mathtt{x}:=\mathtt{x}+1]\!]}{\longmapsto} \big([\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto 0]\big) \stackrel{\overline{\mathcal{A}}[\![\mathtt{\#if}\,(B)\,\mathtt{x}:=1]\!]}{\longmapsto} \big([\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto 1]\big) \end{aligned}$$

After evaluating S_1, the variable `x` has the constant value 1 for all valid configurations. Observe that in the above lifted stores many components are the same, i.e. many configurations have equivalent analysis information. Such lifted stores can be more compactly represented using sharing (e.g., bit vectors or formulae), which in effect will result in more efficient implementation of the lifted analysis.

Let S_2 be a program obtained from S_1, such that `#if` (B) $\mathtt{x} := 1$ is replaced with `#if` (B) $\mathtt{x} := \mathtt{x} - 1$. Then, we have:

$$\overline{\mathcal{A}}[\![S_2]\!]\big([\mathtt{x}\mapsto\top],[\mathtt{x}\mapsto\top],[\mathtt{x}\mapsto\top]\big) = \big([\mathtt{x}\mapsto 0],[\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto -1]\big)$$

We will use programs S_1 and S_2 as running examples throughout the paper. ◀

3 Variability Abstractions

When the set of configurations $\mathbb{K}_\psi$ is large, calculations on the property domain $\mathbb{A}^{\mathbb{K}_\psi}$ become expensive, even if using symbolic representations or sharing to avoid direct storage of $|\mathbb{K}_\psi|$-sized tuples as done in [5]. We want to replace $\mathbb{A}^{\mathbb{K}_\psi}$ with a smaller domain obtained by abstraction and perform an approximate, but feasible, lifted analysis.

---

[1] The functional of the `while` rule is: $\lambda\overline{\Phi}.\,\lambda\overline{a}.\,\overline{a} \mathbin{\dot\sqcup} \overline{\Phi}(\overline{\mathcal{A}}[\![s]\!]\,\overline{a})$.

[2] Since any $k \in \mathbb{K}_\psi$ is a valuation, we have that $k \not\models \theta$ and $k \models \neg\theta$ are equivalent for any $\theta \in \mathit{FeatExp}$.

3.1 Basic Abstractions

We describe a compositional way of constructing abstractions over the domain $\mathbb{A}^{\mathbb{K}}$, where $\mathbb{K}$ represents an arbitrary set of valid configurations, using two basic constructors, join and projection, along with a sequential and parallel composition of abstractions. The set of abstractions *Abs* is generated by the following grammar:

$$\alpha \ ::= \ \boldsymbol{\alpha}^{\text{join}} \mid \boldsymbol{\alpha}^{\text{proj}}_{\varphi} \mid \alpha \circ \alpha \mid \alpha \otimes \alpha \tag{1}$$

where $\varphi \in FeatExp$. Below we define the constructors and motivate them with examples. For readability, we use the constant propagation lattice $\mathbb{A}$ however the results hold for any complete lattice.

Join. Consider the following scenario. An analysis is run interactively, while a developer is typing in a development environment. The analysis finds simple errors and warnings. In this scenario, the analysis must be fast and it should consider all legal configurations $\mathbb{K}$. It is not problematic if some spurious errors are introduced, since, like previously, a more thorough analysis is run regularly. Here, the precision with respect to configurations can be reduced by confounding the control-flow of all the products, obtaining an analysis that runs as if it was analyzing a single product, but involving code variants that participate in all products.

The *join* abstraction gathers the information about all valid configurations $k \in \mathbb{K}$ into one value of $\mathbb{A}$. We formulate the abstraction $\boldsymbol{\alpha}^{\text{join}} : \mathbb{A}^{\mathbb{K}} \to \mathbb{A}^{\{\bigvee_{k\in\mathbb{K}} k\}}$ and the concretization function $\boldsymbol{\gamma}^{\text{join}} : \mathbb{A}^{\{\bigvee_{k\in\mathbb{K}} k\}} \to \mathbb{A}^{\mathbb{K}}$ as follows:

$$\boldsymbol{\alpha}^{\text{join}}(\overline{a}) = \big(\textstyle\bigsqcup_{k\in\mathbb{K}} \pi_k(\overline{a})\big) \qquad \text{and} \qquad \boldsymbol{\gamma}^{\text{join}}(a) = \prod_{k\in\mathbb{K}} a \tag{2}$$

We overload abstraction names (α) to apply not only to domain elements but also to sets of features, sets of configurations, and, later, to program code. The new set of valid configurations is $\boldsymbol{\alpha}^{\text{join}}(\mathbb{K}) = \{\bigvee_{k\in\mathbb{K}} k\}$. Thus, we have only one valid configuration denoted by the formula $\bigvee_{k\in\mathbb{K}} k$. Observe that this means that the obtained abstract domain is effectively $\mathbb{A}^1$, which is isomorphic to $\mathbb{A}$. The proposed abstraction–concretization pair is a Galois connection, which means that it can be used to construct analyses using calculational abstract interpretation:

► **Theorem 2.** $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq}\rangle \underset{\boldsymbol{\alpha}^{\text{join}}}{\overset{\boldsymbol{\gamma}^{\text{join}}}{\leftrightarrows}} \langle \mathbb{A}^{\boldsymbol{\alpha}^{\text{join}}(\mathbb{K})}, \dot{\sqsubseteq}\rangle$ *is a Galois connection* [3] [4].

► **Example 3.** Let us return to the scenario of using *join* for improving analysis performance. Assume that the feature model is given by $\psi = A \vee B$ with valid configurations $\mathbb{K}_\psi = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$. Now, the final stores we obtain by analyzing programs S_1 and S_2 from Example 1 are $\overline{a}_{S_1} = \big([\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto 1]\big)$ and $\overline{a}_{S_2} = \big([\mathtt{x}\mapsto 0],[\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto -1]\big)$. Applying the join abstraction we obtain $\boldsymbol{\alpha}^{\text{join}}(\overline{a}_{S_1}) = \big([\mathtt{x}\mapsto 1]\big)$ and $\boldsymbol{\alpha}^{\text{join}}(\overline{a}_{S_2}) = \big([\mathtt{x}\mapsto \top]\big)$. In both cases the state representation has been significantly decreased. In the former case, the abstraction promptly notices that `x` is a constant regardless of the configuration. In the latter case, the abstraction looses precision by saying that `x` is not a constant in general, even if it was a constant in each of the configurations considered. We will continue using stores $\overline{a}_{S_1}$ and $\overline{a}_{S_2}$ in the subsequent examples. ◄

---

[3] $\langle L, \leq_L\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle M, \leq_M\rangle$ is a *Galois connection* between complete lattices L and M iff α and γ are total functions that satisfy: $\alpha(l) \leq_M m \iff l \leq_L \gamma(m)$ for all $l \in L, m \in M$.

[4] The proofs of all theorems in this section can be found in [14, App. A].

Projection. In industrial practice the number of products actually deployed is often only a small subset of $\mathbb{K}$ [3]. In such case, analyzing all legal (valid) configurations seems unnecessary, and performance of analyses can be improved by abstracting many products away. This is achieved by a configuration projection, which removes configurations that do not satisfy a given constraint, for instance a disjunction of product configurations of interest. Projection can be helpful in other similar scenarios; for instance, to parallelize the analysis—by partitioning the product space using project and analyzing each partition separately.

Let φ be a formula over feature names. We define a *projection* abstraction mapping $\mathbb{A}^{\mathbb{K}}$ into the domain $\mathbb{A}^{\{k\in\mathbb{K}|k\models\varphi\}}$, which preserves only the values corresponding to configurations from $\mathbb{K}$ that satisfy φ. The information about configurations violating φ is disregarded. The abstraction and concretization functions between $\mathbb{A}^{\mathbb{K}}$ and $\mathbb{A}^{\{k\in\mathbb{K}|k\models\varphi\}}$ are defined as follows:

$$\boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\overline{a}) = \prod_{k\in\mathbb{K},k\models\varphi} \pi_k(\overline{a}) \tag{3}$$

$$\boldsymbol{\gamma}^{\text{proj}}_{\varphi}(\overline{a}') = \prod_{k\in\mathbb{K}} \begin{cases} \pi_k(\overline{a}') & \text{if } k \models \varphi \\ \top & \text{if } k \not\models \varphi \end{cases} \tag{4}$$

The new set of configurations is $\boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\mathbb{K}) = \{k \in \mathbb{K} \mid k \models \varphi\}$. Naturally, we also have a Galois connection here:

▶ **Theorem 4.** $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq}\rangle \underset{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}}{\overset{\boldsymbol{\gamma}^{\text{proj}}_{\varphi}}{\leftrightarrows}} \langle \mathbb{A}^{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\mathbb{K})}, \dot{\sqsubseteq}\rangle$ *is a Galois connection.*

Notice that $\boldsymbol{\alpha}^{\text{proj}}_{\text{true}}$ is the identity function, since $k \models$ true for all $k \in \mathbb{K}$. On the other hand $\boldsymbol{\alpha}^{\text{proj}}_{\text{false}}$ is the coarsest collapsing abstraction that maps any tuple into an empty one, since $k \not\models$ false, for all k.

▶ **Example 5.** Let us revisit our scenario, where a set of deployed configurations is much smaller than the set of configurations defined by the feature model ψ. Let us consider the store $\overline{a}_{S_2}$ with the set of valid configurations $\mathbb{K}_{\psi}$ from Example 3. The set of deployed products is defined by formula $\varphi = A$ (so all possible programs with feature A are marketed). By definition of projection (3), we have: $\boldsymbol{\alpha}^{\text{proj}}_{A}(\overline{a}_{S_2}) = \big(\pi_{A\wedge B}(\overline{a}_{S_2}), \pi_{A\wedge\neg B}(\overline{a}_{S_2})\big) = \big([\mathtt{x}\mapsto 0], [\mathtt{x}\mapsto 1]\big)$, and $\boldsymbol{\alpha}^{\text{proj}}_{\neg A}(\overline{a}_{S_2}) = (\pi_{\neg A\wedge B}(\overline{a}_{S_2})) = ([\mathtt{x}\mapsto -1])$. The state representation is effectively decreased to two, respectively one, components. ◀

An attentive reader, might discount the idea of the projection abstraction as being overly heavy. In the end, it appears to be equivalent to running the original analysis, just with a strengthened feature model ($\psi\wedge\varphi$). However, as we shall see in the subsequent developments, projection is indeed useful. Thanks to the composition operators it can enter intricate scenarios, which cannot be expressed using a simple strengthening of a global feature model.

Sequential Composition. We use composition to build complex abstractions out of the basic ones, which also allows us to keep the number of operators in the framework and in the implementation low.

Recall that a composition of two Galois connections is also a Galois connection [11]. Let $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq}\rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \mathbb{A}^{\alpha_1(\mathbb{K})}, \dot{\sqsubseteq}\rangle$ and $\langle \mathbb{A}^{\alpha_1(\mathbb{K})}, \dot{\sqsubseteq}\rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \mathbb{A}^{\alpha_2(\alpha_1(\mathbb{K}))}, \dot{\sqsubseteq}\rangle$ be two Galois connections. Then, we define their *composition* as $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq}\rangle \underset{\alpha_2\circ\alpha_1}{\overset{\gamma_1\circ\gamma_2}{\leftrightarrows}} \langle \mathbb{A}^{(\alpha_2\circ\alpha_1)(\mathbb{K})}, \dot{\sqsubseteq}\rangle$, where

$$(\alpha_2 \circ \alpha_1)(\overline{a}) = \alpha_2(\alpha_1(\overline{a})) \quad \text{and} \quad (\gamma_1 \circ \gamma_2)(\overline{a}') = \gamma_1(\gamma_2(\overline{a}')) \tag{5}$$

for $\overline{a} \in \mathbb{A}^{\mathbb{K}}$ and $\overline{a}' \in \mathbb{A}^{(\alpha_2\circ\alpha_1)(\mathbb{K})}$. Also $(\alpha_2 \circ \alpha_1)(\mathbb{K}) = \alpha_2(\alpha_1(\mathbb{K}))$.

▶ **Example 6.** Now consider the process of deriving an analysis, which only considers products actually deployed described by a formula φ (see previous example), but which should trade precision for speed, by confounding their execution. Such an analysis is derived using the composed abstraction: $\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\varphi}$.

Let $\varphi = A$. Configurations $A \wedge B$ and $A \wedge \neg B$ satisfy φ, whereas $\neg\varphi$ is satisfied only by $\neg A \wedge B$. We have: $\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}(\overline{a}_{S_2}) = (\pi_{A\wedge B}(\overline{a}_{S_2}) \sqcup \pi_{A\wedge\neg B}(\overline{a}_{S_2})) = ([\mathtt{x} \mapsto \top])$ and $\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\neg A}(\overline{a}_{S_2}) = (\pi_{\neg A\wedge B}(\overline{a}_{S_2})) = ([\mathtt{x} \mapsto -1])$. ◀

Parallel Composition. Consider a product line where two disjoint groups of products share the same code base: one group is correctness critical, the other comprises correctness non-critical products. The former should be analyzed with highest precision possible to obtain the most precise analysis results, the latter can be analyzed faster. We can set up such analyses by using a projection abstraction to analyze the correctness critical group precisely, and the join abstraction to analyze the non-critical group. However running the analyses twice, ignores the fact that the code is shared between the groups. We can combine two separate analyses by creating a compound abstraction: a *product* of the two. The product abstraction will correspond exactly to executing the projection on the correctness critical products, and join on the non-critical ones. But since the product creates a single Galois connection of the two, it can be used to derive an analysis which will deliver this in a single run, which is more efficient overall, due to reuse of the states explored.

Galois connections $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \mathbb{A}^{\alpha_1(\mathbb{K})}, \dot{\sqsubseteq} \rangle$ and $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \mathbb{A}^{\alpha_2(\mathbb{K})}, \dot{\sqsubseteq} \rangle$ over the same domain $\mathbb{A}^{\mathbb{K}}$ can be composed into one that combines the abstraction results "side-by-side". The result is a new compound abstraction, $\alpha_1 \otimes \alpha_2$, of the domain $\mathbb{A}^{\mathbb{K}}$ obtained by applying the two simpler abstractions in parallel. The parallel composition of abstractions is defined using a direct tensor product. For the resulting Galois connection, we have $\alpha_1 \otimes \alpha_2(\mathbb{K}) = \alpha_1(\mathbb{K}) \cup \alpha_2(\mathbb{K})$. Given $\overline{a}_1 \in \mathbb{A}^{\alpha_1(\mathbb{K})}$ and $\overline{a}_2 \in \mathbb{A}^{\alpha_2(\mathbb{K})}$, we first define $\overline{a}_1 \times \overline{a}_2 \in \alpha_1(\mathbb{K}) \cup \alpha_2(\mathbb{K})$ as:

$$\overline{a}_1 \times \overline{a}_2 = \prod_{k \in \alpha_1(\mathbb{K}) \cup \alpha_2(\mathbb{K})} \begin{cases} \pi_k(\overline{a}_1) & \text{if } k \in \alpha_1(\mathbb{K}) \setminus \alpha_2(\mathbb{K}) \\ \pi_k(\overline{a}_1) \sqcup \pi_k(\overline{a}_2) & \text{if } k \in \alpha_1(\mathbb{K}) \cap \alpha_2(\mathbb{K}) \\ \pi_k(\overline{a}_2) & \text{if } k \in \alpha_2(\mathbb{K}) \setminus \alpha_1(\mathbb{K}) \end{cases} \tag{6}$$

The direct tensor product is given as $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\alpha_1 \otimes \alpha_2}{\overset{\gamma_1 \otimes \gamma_2}{\leftrightarrows}} \langle \mathbb{A}^{(\alpha_1 \otimes \alpha_2)(\mathbb{K})}, \dot{\sqsubseteq} \rangle$, where

$$(\alpha_1 \otimes \alpha_2)(\overline{a}) = \alpha_1(\overline{a}) \times \alpha_2(\overline{a}) \tag{7}$$

$$(\gamma_1 \otimes \gamma_2)(\overline{a}') = \gamma_1(\pi_{\alpha_1(\mathbb{K})}(\overline{a}')) \sqcap \gamma_2(\pi_{\alpha_2(\mathbb{K})}(\overline{a}')) \ \text{, where} \tag{8}$$

$\pi_{\alpha_1(\mathbb{K})}(\overline{a}') = \prod_{k \in \alpha_1(\mathbb{K})} \pi_k(\overline{a}')$ and $\pi_{\alpha_2(\mathbb{K})}(\overline{a}') = \prod_{k \in \alpha_2(\mathbb{K})} \pi_k(\overline{a}')$, for $\overline{a}' \in \mathbb{A}^{(\alpha_1 \otimes \alpha_2)(\mathbb{K})}$.

▶ **Theorem 7.** $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\alpha_1 \otimes \alpha_2}{\overset{\gamma_1 \otimes \gamma_2}{\leftrightarrows}} \langle \mathbb{A}^{(\alpha_1 \otimes \alpha_2)(\mathbb{K})}, \dot{\sqsubseteq} \rangle$ *is a Galois connection.*

▶ **Example 8.** Let us assume that for products with feature A we need precise analysis results, and for products without this feature we do not need so precise results. We are interested in analyzing products with A thoroughly, while the analysis of the products without A can be speeded up. To this end we build the following abstraction: $\boldsymbol{\alpha}^{\text{proj}}_{A} \otimes (\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\neg A})$. ◀

3.2 Derived Abstractions

We shall now discuss three more abstractions that can be derived from the above basic constructors.

Join-Project. Recall the construction of Example 6, where we combined projection with a join in order to confound a subset of legal configurations. This pattern has occurred so often in our exercises that we introduced a syntactic sugar for it. For a formula φ over features, the abstraction $\boldsymbol{\alpha}^{\text{join}}_{\varphi}$ gathers the information about all valid configurations $k \in \mathbb{K}$ that satisfy φ, i.e. $k \models \varphi$, into one value of $\mathbb{A}$, whereas the information about all other valid configurations $k \in \mathbb{K}$ that do not satisfy φ is disregarded. We define

$$\boldsymbol{\alpha}^{\text{join}}_{\varphi} = \boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\varphi} \quad \text{and} \quad \boldsymbol{\gamma}^{\text{join}}_{\varphi} = \boldsymbol{\gamma}^{\text{proj}}_{\varphi} \circ \boldsymbol{\gamma}^{\text{join}} \tag{9}$$

where $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}}{\overset{\boldsymbol{\gamma}^{\text{proj}}_{\varphi}}{\leftrightarrows}} \langle \mathbb{A}^{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\mathbb{K})}, \dot{\sqsubseteq} \rangle$ and $\langle \mathbb{A}^{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\mathbb{K})}, \dot{\sqsubseteq} \rangle \underset{\boldsymbol{\alpha}^{\text{join}}}{\overset{\boldsymbol{\gamma}^{\text{join}}}{\leftrightarrows}} \langle \mathbb{A}^{(\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\varphi})(\mathbb{K}))}, \dot{\sqsubseteq} \rangle$ are Galois connections. Now the compositions in Example 6 can be written simply as $\boldsymbol{\alpha}^{\text{join}}_{A}$ and $\boldsymbol{\alpha}^{\text{join}}_{\neg A}$.

Ignoring features. Consider a scenario, where a configurable third-party component is integrated into a product line. The code base is large, and a static analysis does not scale to this size. In a compile-analyze-test cycle errors appear most often in the newly written code, and are thus relatively little influenced by how the features of the third party component are configured. Lowering precision on analyzing external components can allow finding errors faster. This scenario can be realized using a feature projection, which simplifies feature domains by confounding executions differing only on uninteresting features.

Before defining feature projection, let us consider a simpler case of ignoring a single feature $A \in \mathbb{F}$ that is not directly relevant for current analysis. The *ignore feature* abstraction merges any configurations that only differ with regard to A, and are identical with regard to remaining features, $\mathbb{F} \backslash \{A\}$. We write $k \backslash_A$ for a formula obtained by eliminating the feature A from k. The new set of configurations is given by $\boldsymbol{\alpha}^{\text{fignore}}_{A}(\mathbb{K}) = \{\bigvee_{k \in \mathbb{K}, k\backslash_A \equiv k'} k \mid k' \in \{k\backslash_A \mid k \in \mathbb{K}\}\}$. The abstraction $\boldsymbol{\alpha}^{\text{fignore}}_{A} : \mathbb{A}^{\mathbb{K}} \to \mathbb{A}^{\boldsymbol{\alpha}^{\text{fignore}}_{A}(\mathbb{K})}$ and concretization functions $\boldsymbol{\gamma}^{\text{fignore}}_{A} : \mathbb{A}^{\boldsymbol{\alpha}^{\text{fignore}}_{A}(\mathbb{K})} \to \mathbb{A}^{\mathbb{K}}$ are:

$$\boldsymbol{\alpha}^{\text{fignore}}_{A}(\overline{a}) = \prod_{k' \in \boldsymbol{\alpha}^{\text{fignore}}_{A}(\mathbb{K})} \bigsqcup_{k \in \mathbb{K}, k \models k'} \pi_k(\overline{a}) \tag{10}$$

$$\boldsymbol{\gamma}^{\text{fignore}}_{A}(\overline{a}') = \prod_{k \in \mathbb{K}} \pi_{k'}(\overline{a}') \;\; \text{if } k \models k' \tag{11}$$

It turns out that ignoring features can be derived from the above basic abstractions as shown in the following theorem:

▶ **Theorem 9.** *Let* $\boldsymbol{\alpha}^{\text{fignore}}_{A}(\mathbb{K}) = \{k'_1, \ldots, k'_n\}$. *Then:*

$$\boldsymbol{\alpha}^{\text{fignore}}_{A} = \boldsymbol{\alpha}^{\text{join}}_{k'_1} \otimes \cdots \otimes \boldsymbol{\alpha}^{\text{join}}_{k'_n} \quad \textit{and} \quad \boldsymbol{\gamma}^{\text{fignore}}_{A} = \boldsymbol{\gamma}^{\text{join}}_{k'_1} \otimes \cdots \otimes \boldsymbol{\gamma}^{\text{join}}_{k'_n} \;.$$

▶ **Example 10.** We consider the lifted store $\overline{a}_{S_2}$ with $\mathbb{K}_\psi = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$. Then, we have $\boldsymbol{\alpha}^{\text{fignore}}_{A}(\mathbb{K}_\psi) = \{(A \wedge B) \vee (\neg A \wedge B), A \wedge \neg B\}$ and $\boldsymbol{\alpha}^{\text{fignore}}_{A}(\overline{a}_{S_2}) = (\pi_{A\wedge B}(\overline{a}_{S_2}) \sqcup \pi_{\neg A\wedge B}(\overline{a}_{S_2}), \pi_{A\wedge\neg B}(\overline{a}_{S_2})) = ([\mathtt{x} \mapsto \top], [\mathtt{x} \mapsto 1])$. On the other hand, we have $\boldsymbol{\alpha}^{\text{fignore}}_{B}(\mathbb{K}_\psi) = \{(A \wedge B) \vee (A \wedge \neg B), \neg A \wedge B\}$ and $\boldsymbol{\alpha}^{\text{fignore}}_{A}(\overline{a}_{S_2}) = (\pi_{A\wedge B}(\overline{a}_{S_2}) \sqcup \pi_{A\wedge\neg B}(\overline{a}_{S_2}), \pi_{\neg A\wedge B}(\overline{a}_{S_2})) = ([\mathtt{x} \mapsto \top], [\mathtt{x} \mapsto -1])$. ◀

Feature Projection. Now, if we need to ignore a larger number of features (say features outside a certain component of interest), we can do it using a feature projection operator which simply ignores a set of features $\{A_1, \ldots, A_k\} \subseteq \mathbb{F}$:

$$\boldsymbol{\alpha}^{\text{fproj}}_{\{A_1,\ldots,A_k\}} = \boldsymbol{\alpha}^{\text{fignore}}_{A_1} \circ \cdots \circ \boldsymbol{\alpha}^{\text{fignore}}_{A_k} \quad \text{and} \quad \boldsymbol{\gamma}^{\text{fproj}}_{\{A_1,\ldots,A_k\}} = \boldsymbol{\gamma}^{\text{fignore}}_{A_k} \circ \cdots \circ \boldsymbol{\gamma}^{\text{fignore}}_{A_1}$$

$$
\begin{aligned}
&(\alpha \circ \overline{\mathcal{A}}[\![\texttt{\#if}\ (\theta)\ s]\!] \circ \gamma)(\overline{d}) = \alpha(\overline{\mathcal{A}}[\![\texttt{\#if}\ (\theta)\ s]\!](\gamma(\overline{d}))) = && \text{(by def. of } \circ\text{)} \\
&= \alpha\Big(\prod_{k \in \mathbb{K}_\psi} \begin{cases} \pi_k(\overline{\mathcal{A}}[\![s]\!]\gamma(\overline{d})) & \text{if } k \models \theta \\ \pi_k(\gamma(\overline{d})) & \text{if } k \not\models \theta \end{cases} \Big) && \text{(def. of } \overline{\mathcal{A}} \text{ in Fig. 2)} \\
&\dot\sqsubseteq \prod_{k' \in \alpha(\mathbb{K}_\psi)} \begin{cases} \pi_{k'}(\alpha(\overline{\mathcal{A}}[\![s]\!]\gamma(\overline{d}))) & \text{if } k' \models \theta \\ \pi_{k'}(\alpha(\gamma(\overline{d}))) \sqcup \pi_{k'}(\alpha(\overline{\mathcal{A}}[\![s]\!]\gamma(\overline{d}))) & \text{if } \mathrm{sat}(k' \wedge \theta) \wedge \mathrm{sat}(k' \wedge \neg\theta) \\ \pi_{k'}(\alpha(\gamma(\overline{d}))) & \text{if } k' \models \neg\theta \end{cases} \\
& && \text{(by Lemma 2 in [14, App. C])} \\
&\dot\sqsubseteq \prod_{k' \in \alpha(\mathbb{K}_\psi)} \begin{cases} \pi_{k'}(\overline{\mathcal{D}}_\alpha[\![s]\!]\overline{d}) & \text{if } k' \models \theta \\ \pi_{k'}(\overline{d}) \sqcup \pi_{k'}(\overline{\mathcal{D}}_\alpha[\![s]\!]\overline{d}) & \text{if } \mathrm{sat}(k' \wedge \theta) \wedge \mathrm{sat}(k' \wedge \neg\theta) \\ \pi_{k'}(\overline{d}) & \text{if } k' \models \neg\theta \end{cases} && \text{(by IH and } \alpha \circ \gamma \text{ reductive)} \\
&= \overline{\mathcal{D}}_\alpha[\![\texttt{\#if}\ (\theta)\ s]\!]\,\overline{d}
\end{aligned}
$$

Figure 3 Calculational derivation of $\overline{\mathcal{D}}_\alpha[\![\texttt{\#if}\ (\theta)\ s]\!]$, the abstraction of $\overline{\mathcal{A}}[\![\texttt{\#if}\ (\theta)\ s]\!]$. The 'reductive' property of all Galois connections is $(\alpha \circ \gamma)(\overline{d}) \sqsubseteq \overline{d}$ for all $\overline{d}$.

It follows from the theorems of Section 3.1 that all the derived pairs of abstraction and concretization functions are Galois connections.

4 Abstracting Lifted Analyses

We will now demonstrate how to derive abstracted lifted analyses using the operators of Section 3, using the case of constant propagation for $\overline{\text{IMP}}$ programs as an example. Recall that this analysis has been specified by: 1) the domain $\mathbb{A}^{\mathbb{K}_\psi}$; 2) the statement transfer function $\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\mathbb{K}_\psi}$; and 3) the expression evaluation function $\overline{\mathcal{A}'}[\![e]\!] : (\mathbb{A} \to \mathit{Const})^{\mathbb{K}_\psi}$. Let $\langle \mathbb{A}^{\mathbb{K}_\psi}, \dot\sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\longleftrightarrow}} \langle \mathbb{A}^{\alpha(\mathbb{K}_\psi)}, \dot\sqsubseteq \rangle$ be a Galois connection constructed using the abstractions presented in Section 3. We will also write $(\alpha, \gamma) \in \mathit{Abs}$ to denote a Galois connection obtained in such way.

Any function f defined on the concrete domain of a Galois connection can be abstracted to work on the abstract domain by applying concretization to its argument and an abstraction to its value, i.e. by the function $F = \alpha \circ f \circ \gamma$, where $\circ$ denotes the usual composition of functions. In fact, any monotone over-approximation of the composition $\alpha \circ f \circ \gamma$ is sufficient for a sound analysis. Even fixed points can be transferred from a concrete to an abstract domain of a Galois connection. If both domains are complete lattices and f is a monotone function on the concrete domain, then by the fixed point transfer theorem (FPT for short) [10]: $\alpha(\mathrm{lfp} f) \sqsubseteq \mathrm{lfp} F \sqsubseteq \mathrm{lfp} F^\#$. Here $F = \alpha \circ f \circ \gamma$ and $F^\#$ is some monotone, conservative *over*-approximation of F, i.e. $F \sqsubseteq F^\#$. The calculational approach to abstract interpretation [9] used in this work, advocates simple algebraic manipulation to obtain a *direct expression* for the function F (if it exists) or for an over-approximation $F^\#$.

In our case, for any lifted store $\overline{a} \in \mathbb{A}^{\mathbb{K}_\psi}$, we calculate an abstracted lifted store by $\alpha(\overline{a}) = \overline{d} \in \mathbb{A}^{\alpha(\mathbb{K}_\psi)}$. Now, we use a Galois connection to derive an over-approximation of $\alpha \circ \overline{\mathcal{A}}[\![s]\!] \circ \gamma$ obtaining a new abstracted statement transfer function $\overline{\mathcal{D}}_\alpha[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\alpha(\mathbb{K}_\psi)}$. Similarly, one can derive an abstracted analysis for expressions $\overline{\mathcal{D}'}_\alpha[\![e]\!]$, approximating $\alpha \circ \overline{\mathcal{A}'}[\![e]\!] \circ \gamma$. These approximations are derived using structural induction on statements (respectively on expressions), in a process that resembles a simple algebraic calculation,

$$
\begin{aligned}
&(\alpha \circ \overline{\mathcal{A}'}[\![e_0 \oplus e_1]\!] \circ \gamma)(\overline{d}) \\
&= \alpha\big(\prod_{k\in\mathbb{K}_\psi} \pi_k(\overline{\mathcal{A}'}[\![e_0]\!]\gamma(\overline{d})) \mathbin{\widehat{\oplus}} \pi_k(\overline{\mathcal{A}'}[\![e_1]\!]\gamma(\overline{d}))\big) && \text{(by def. of } \circ\text{, and } \overline{\mathcal{A}'} \text{ in Fig. 2)} \\
&= \alpha\big(\prod_{k\in\mathbb{K}_\psi} \pi_k(\overline{\mathcal{A}'}[\![e_0]\!]\gamma(\overline{d}) \mathbin{\dot{\widehat{\oplus}}} \overline{\mathcal{A}'}[\![e_1]\!]\gamma(\overline{d}))\big) && \text{(by def. of } \pi_k \text{ and } \dot{\widehat{\oplus}}) \\
&= \prod_{k'\in\alpha(\mathbb{K}_\psi)} \pi_{k'}(\alpha\big(\overline{\mathcal{A}'}[\![e_0]\!]\gamma(\overline{d}) \mathbin{\dot{\widehat{\oplus}}} \overline{\mathcal{A}'}[\![e_1]\!]\gamma(\overline{d})\big)) && \text{(by def. of } \alpha) \\
&\dot{\sqsubseteq} \prod_{k'\in\alpha(\mathbb{K}_\psi)} \pi_{k'}(\alpha(\overline{\mathcal{A}'}[\![e_0]\!]\gamma(\overline{d})) \mathbin{\dot{\widehat{\oplus}}} \alpha(\overline{\mathcal{A}'}[\![e_1]\!]\gamma(\overline{d}))) && \text{(by Lemma 3 in [14, App. C])} \\
&\dot{\sqsubseteq} \prod_{k'\in\alpha(\mathbb{K}_\psi)} \pi_{k'}(\overline{\mathcal{D}'}_\alpha[\![e_0]\!]\overline{d} \mathbin{\dot{\widehat{\oplus}}} \overline{\mathcal{D}'}_\alpha[\![e_1]\!]\overline{d}) && \text{(by IH, twice)} \\
&\dot{\sqsubseteq} \prod_{k'\in\alpha(\mathbb{K}_\psi)} \pi_{k'}(\overline{\mathcal{D}'}_\alpha[\![e_0]\!]\overline{d}) \mathbin{\widehat{\oplus}} \pi_{k'}(\overline{\mathcal{D}'}_\alpha[\![e_1]\!]\overline{d}) && \text{(by def. of } \pi_{k'} \text{ and } \dot{\widehat{\oplus}}) \\
&= \overline{\mathcal{D}'}_\alpha[\![e_0 \oplus e_1]\!]\overline{d}
\end{aligned}
$$

Figure 4 Calculational derivation of $\overline{\mathcal{D}}_\alpha[\![e_0 \oplus e_1]\!]$.

deceivingly akin to equation reasoning.

Let us consider the derivation steps for the static conditional statement (`#if` (θ) s) in detail. Our inductive hypothesis (IH) is that for statements s' that are structurally smaller than (`#if` (θ) s) the (yet-to-be-calculated) $\overline{\mathcal{D}}_\alpha[\![s']\!]$ soundly approximates $\alpha \circ \overline{\mathcal{A}}[\![s']\!] \circ \gamma$, formally: $\alpha \circ \overline{\mathcal{A}}[\![s']\!] \circ \gamma \mathrel{\dot{\sqsubseteq}} \overline{\mathcal{D}}_\alpha[\![s']\!]$. The derivation in Fig. 3 begins with composing the concretization and abstraction functions with the concrete transfer function and then proceeds by expanding definitions. An (inner) induction on the structure of the abstraction α follows, delegated to the Appendix for brevity. In the last step we apply the inductive hypothesis, to obtain a closed representation independent of $\overline{\mathcal{A}}$. This representation, just before the final equality, is the newly obtained (calculated) definition of the abstracted analysis $\overline{\mathcal{D}}_\alpha$. Interestingly, the derivation is independent of the structure of the abstraction α, so this form works for any abstraction specified using our operators. We give derivational steps for $e_0 \oplus e_1$ in Fig. 4.

The derivations for other cases are similar and can be found in [14, App. B]. The process results in the definitions of $\overline{\mathcal{D}}_\alpha[\![s]\!]$ and $\overline{\mathcal{D}'}_\alpha[\![e]\!]$ presented in Fig. 5. Soundness of the abstracted analysis follows by construction; more precisely the complete calculation constitutes an inductive proof of the following theorem:

▶ **Theorem 11** (Soundness of Abstracted Analysis). *We have that:*

(i) $\forall e \in \mathit{Exp}, (\alpha, \gamma) \in \mathit{Abs}, \overline{d} \in \mathbb{A}^{\alpha(\mathbb{K}_\psi)} : \ \alpha \circ \overline{\mathcal{A}'}[\![e]\!] \circ \gamma(\overline{d}) \mathrel{\dot{\sqsubseteq}} \overline{\mathcal{D}'}_\alpha[\![e]\!]\,\overline{d}$
(ii) $\forall s \in \mathit{Stm}, (\alpha, \gamma) \in \mathit{Abs}, \overline{d} \in \mathbb{A}^{\alpha(\mathbb{K}_\psi)} : \ \alpha \circ \overline{\mathcal{A}}\,[\![s]\!] \circ \gamma(\overline{d}) \mathrel{\dot{\sqsubseteq}} \overline{\mathcal{D}}_\alpha[\![s]\!]\,\overline{d}$

Monotonicity of the abstracted analysis is shown in [14, App. D].

▶ **Theorem 12** (Monotonicity of Abstracted Analysis). *For all $s \in \mathit{Stm}$ and $e \in \mathit{Exp}$, $\overline{\mathcal{D}}_\alpha[\![s]\!]$ and $\overline{\mathcal{D}'}_\alpha[\![e]\!]$ are monotone functions.*

▶ **Example 13.** Consider the program S_1 from Example 1, with $\mathbb{K}_\psi = \{A{\wedge}B, A{\wedge}\neg B, \neg A{\wedge}B\}$. We calculate $\overline{\mathcal{D}}_{\alpha_1}[\![S_1]\!]$ for $\alpha_1 = \boldsymbol{\alpha}_A^{\text{join}}$. Following the rules of Fig. 5, we obtain the following confounded abstract execution off all configurations containing the feature A:

$$
\big([\mathtt{x} \mapsto \top]\big) \overset{\overline{\mathcal{D}}_{\alpha_1}[\![\mathtt{x:=0}]\!]}{\longmapsto} \big([\mathtt{x} \mapsto 0]\big) \overset{\overline{\mathcal{D}}_{\alpha_1}[\![\mathtt{\#if}\ (A)\ \mathtt{x:=x+1}]\!]}{\longmapsto} \big([\mathtt{x} \mapsto 1]\big) \overset{\overline{\mathcal{D}}_{\alpha_1}[\![\mathtt{\#if}\ (B)\ \mathtt{x:=1}]\!]}{\longmapsto} \big([\mathtt{x} \mapsto 1]\big)
$$

$$
\begin{aligned}
\overline{\mathcal{D}}_\alpha[\![\texttt{skip}]\!] &= \lambda\overline{d}.\,\overline{d} \\
\overline{\mathcal{D}}_\alpha[\![\texttt{x := } e]\!] &= \lambda\overline{d}.\prod_{k'\in\alpha(\mathbb{K}_\psi)} (\pi_{k'}(\overline{d}))[\texttt{x} \mapsto \pi_{k'}(\overline{\mathcal{D}'}_\alpha[\![e]\!]\overline{d})] \\
\overline{\mathcal{D}}_\alpha[\![s_0\ \texttt{;}\ s_1]\!] &= \overline{\mathcal{D}}_\alpha[\![s_1]\!] \circ \overline{\mathcal{D}}_\alpha[\![s_0]\!] \\
\overline{\mathcal{D}}_\alpha[\![\texttt{if}\ e\ \texttt{then}\ s_0\ \texttt{else}\ s_1]\!] &= \lambda\overline{d}.\,\overline{\mathcal{D}}_\alpha[\![s_0]\!]\overline{d} \,\dot\sqcup\, \overline{\mathcal{D}}_\alpha[\![s_1]\!]\overline{d} \\
\overline{\mathcal{D}}_\alpha[\![\texttt{while}\ e\ \texttt{do}\ s]\!] &= \mathrm{lfp}\lambda\overline{\Phi}.\,\lambda\overline{d}.\,\overline{d} \,\dot\sqcup\, \overline{\Phi}(\overline{\mathcal{D}}_\alpha[\![s]\!]\,\overline{d}) \\
\overline{\mathcal{D}}_\alpha[\![\texttt{\#if}\ (\theta)\ s]\!] &= \lambda\overline{d}.\prod_{k'\in\alpha(\mathbb{K}_\psi)} \begin{cases} \pi_{k'}(\overline{\mathcal{D}}_\alpha[\![s]\!]\overline{d}) & \text{if } k' \models \theta \\ \pi_{k'}(\overline{d}) \sqcup \pi_{k'}(\overline{\mathcal{D}}_\alpha[\![s]\!]\overline{d}) & \text{if } \mathrm{sat}(k'\wedge\theta) \wedge \mathrm{sat}(k'\wedge\neg\theta) \\ \pi_{k'}(\overline{d}) & \text{if } k' \models \neg\theta \end{cases} \\
\overline{\mathcal{D}'}_\alpha[\![n]\!] &= \lambda\overline{d}.\prod_{k'\in\alpha(\mathbb{K}_\psi)} \texttt{n} \\
\overline{\mathcal{D}'}_\alpha[\![\texttt{x}]\!] &= \lambda\overline{d}.\prod_{k'\in\alpha(\mathbb{K}_\psi)} \pi_{k'}(\overline{d})(\texttt{x}) \\
\overline{\mathcal{D}'}_\alpha[\![e_0 \oplus e_1]\!] &= \lambda\overline{d}.\prod_{k'\in\alpha(\mathbb{K}_\psi)} \pi_{k'}(\overline{\mathcal{D}'}_\alpha[\![e_0]\!]\overline{d}) \mathbin{\widehat{\oplus}} \pi_{k'}(\overline{\mathcal{D}'}_\alpha[\![e_1]\!]\overline{d})
\end{aligned}
$$

Figure 5 Definitions of $\overline{\mathcal{D}}_\alpha[\![\overline{s}]\!] : (\mathbb{A} \to \mathbb{A})^{\alpha(\mathbb{K}_\psi)}$ and $\overline{\mathcal{D}'}_\alpha[\![\overline{e}]\!] : (\mathbb{A} \to \mathit{Const})^{\alpha(\mathbb{K}_\psi)}$.

In the last step we used $\overline{\mathcal{D}}_{\alpha_1}[\![\texttt{\#if}\,(B)\ \texttt{x} := 1]\!]([\texttt{x} \mapsto 1]) = ([\texttt{x} \mapsto 1]) \,\dot\sqcup\, \overline{\mathcal{D}}_{\alpha_1}[\![\texttt{x} := 1]\!]([\texttt{x} \mapsto 1])$ since $((A \wedge B) \vee (A \wedge \neg B)) \wedge B$ and $((A \wedge B) \vee (A \wedge \neg B)) \wedge \neg B$ are both satisfiable. The final result shows that the value of x is the constant 1 for every configuration that satisfies A. On the other hand, for the program S_2 and the same abstraction we obtain $\overline{\mathcal{D}}_{\alpha_1}[\![S_2]\!]([\texttt{x} \mapsto \top]) = ([\texttt{x} \mapsto \top])$, so the value of x is lost (approximated) by $\overline{\mathcal{D}}_{\alpha_1}$. ◀

We may implement the abstracted analysis in Fig. 5 directly by using Kleene's fixed point theorem to calculate fixed points of loops iteratively. But, we can also extract corresponding data-flow equations, and then apply the known iterative algorithms to calculate fixed-point solutions. We assume that the individual statements are uniquely labelled with labels ℓ. Given an abstraction α, for each statement s^ℓ we generate two abstracted stores $[\![s^\ell]\!]^{\alpha}_{\mathsf{in}}, [\![s^\ell]\!]^{\alpha}_{\mathsf{out}} : \mathbb{A}^{\alpha(\mathbb{K}_\psi)}$, which describe the input and output abstract store for all configurations before and after executing the statement s^ℓ. They are related with the definitions for abstracted analysis $\overline{\mathcal{D}}_\alpha$ given in Fig. 5 as follows: for each statement s the input store $[\![s^\ell]\!]^{\alpha}_{\mathsf{in}}$ is substituted for the parameter $\overline{d}$, and the output store $[\![s^\ell]\!]^{\alpha}_{\mathsf{out}}$ for the value of the corresponding function. The complete list of data-flow equations are given in Fig 6. We can derive data-flow equations for expressions as well, but for brevity we refer directly to $\overline{\mathcal{D}'}_\alpha[\![e]\!]$ function. The obtained data-flow equations are provably sound as shown in [14, App. E].

▶ **Theorem 14** (Soundness of Abstracted Data-Flow Equations)**.** *For all $s \in \mathit{Stm}$ and $\alpha \in \mathit{Abs}$, such that $[\![s^\ell]\!]^{\alpha}_{\mathsf{in}}$ and $[\![s^\ell]\!]^{\alpha}_{\mathsf{out}}$ satisfy the data-flow equations in Fig. 6, it holds:*

$$\overline{\mathcal{D}}_\alpha[\![s^\ell]\!]([\![s^\ell]\!]^{\alpha}_{\mathsf{in}}) \ \dot\sqsubseteq\ [\![s^\ell]\!]^{\alpha}_{\mathsf{out}}$$

5 Variability Abstraction with Syntactic Transformation

The analyses $\overline{\mathcal{A}}$ and $\overline{\mathcal{D}}_\alpha$ can be implemented either directly by using definitions of Figs. 2 and 5, or by extracting the corresponding data-flow equations. An entirely different way

$$
\begin{aligned}
\llbracket \texttt{skip}^{\ell} \rrbracket^{\alpha}_{\text{out}} &= \llbracket \texttt{skip}^{\ell} \rrbracket^{\alpha}_{\text{in}} \\
\underline{\forall k' \in \alpha(\mathbb{K}_\psi)}\colon \pi_{k'}(\llbracket \texttt{x :=}^{\ell}\ e^{\ell 0} \rrbracket^{\alpha}_{\text{out}}) &= \pi_{k'}(\llbracket \texttt{x :=}^{\ell}\ e^{\ell 0} \rrbracket_{\overline{\text{in}}})^{\alpha}[\texttt{x} \mapsto \pi_{k'}(\overline{\mathcal{D}'}_{\alpha}\llbracket e^{\ell 0}\rrbracket \llbracket \texttt{x :=}^{\ell}\ e^{\ell 0} \rrbracket^{\alpha}_{\text{in}})] \\
\llbracket s_0^{\ell 0}\ ;^{\ell}\ s_1^{\ell 1} \rrbracket^{\alpha}_{\text{out}} &= \llbracket s_1^{\ell 1} \rrbracket^{\alpha}_{\text{out}} \\
\llbracket s_1^{\ell 1} \rrbracket^{\alpha}_{\text{in}} &= \llbracket s_0^{\ell 0} \rrbracket^{\alpha}_{\text{out}} \\
\llbracket s_0^{\ell 0} \rrbracket^{\alpha}_{\text{in}} &= \llbracket s_0^{\ell 0}\ ;^{\ell}\ s_1^{\ell 1} \rrbracket^{\alpha}_{\text{in}} \\
\llbracket \texttt{if}^{\ell}\ e\ \texttt{then}\ s_0^{\ell 0}\ \texttt{else}\ s_1^{\ell 1} \rrbracket^{\alpha}_{\text{out}} &= \llbracket s_0^{\ell 0} \rrbracket^{\alpha}_{\text{out}} \dot\sqcup \llbracket s_1^{\ell 1} \rrbracket^{\alpha}_{\text{out}} \\
\llbracket s_0^{\ell 0} \rrbracket^{\alpha}_{\text{in}} &= \llbracket \texttt{if}^{\ell}\ e\ \texttt{then}\ s_0^{\ell 0}\ \texttt{else}\ s_1^{\ell 1} \rrbracket^{\alpha}_{\text{in}} \\
\llbracket s_1^{\ell 1} \rrbracket^{\alpha}_{\text{in}} &= \llbracket \texttt{if}^{\ell}\ e\ \texttt{then}\ s_0^{\ell 0}\ \texttt{else}\ s_1^{\ell 1} \rrbracket^{\alpha}_{\text{in}} \\
\llbracket \texttt{while}^{\ell}\ e\ \texttt{do}\ s^{\ell 0} \rrbracket^{\alpha}_{\text{out}} &= \llbracket s^{\ell 0} \rrbracket^{\alpha}_{\text{in}} \\
\llbracket s^{\ell 0} \rrbracket^{\alpha}_{\text{in}} &= \llbracket \texttt{while}^{\ell}\ e\ \texttt{do}\ s^{\ell 0} \rrbracket^{\alpha}_{\text{in}} \dot\sqcup \llbracket s^{\ell 0} \rrbracket^{\alpha}_{\text{out}} \\
\underline{\forall k' \in \alpha(\mathbb{K}_\psi)}\colon \pi_{k'}(\llbracket \texttt{\#if}^{\ell}\ (\theta)\ s^{\ell 0} \rrbracket^{\alpha}_{\text{out}}) &= \begin{cases} \pi_{k'}(\llbracket s^{\ell 0} \rrbracket^{\alpha}_{\text{out}}) & \text{if } k' \models \theta \\ \pi_{k'}(\llbracket \texttt{\#if}^{\ell}\ (\theta)\ s^{\ell 0} \rrbracket^{\alpha}_{\text{in}}) \sqcup \pi_{k'}(\llbracket s^{\ell 0} \rrbracket^{\alpha}_{\text{out}}) & \text{if sat}(k' \wedge \theta) \wedge \text{sat}(k' \wedge \neg\theta) \\ \pi_{k'}(\llbracket \texttt{\#if}^{\ell}\ (\theta)\ s^{\ell 0} \rrbracket^{\alpha}_{\text{in}}) & \text{if } k' \models \neg\theta \end{cases} \\
\underline{\forall k' \in \alpha(\mathbb{K}_\psi)}\colon \pi_{k'}(\llbracket s^{\ell 0} \rrbracket^{\alpha}_{\text{in}}) &= \pi_{k'}(\llbracket \texttt{\#if}^{\ell}\ (\theta)\ s^{\ell 0} \rrbracket^{\alpha}_{\text{in}}) \quad \text{if sat}(k' \wedge \theta)
\end{aligned}
$$

Figure 6 Data-flow equations for abstracted constant propagation.

to implement $\overline{\mathcal{D}}_{\alpha}$ is to execute the abstraction on the source program, before running the analysis, and then running the previously existing analysis $\overline{\mathcal{A}}$ on this transformed program. We take this route as it allows to completely reuse the effort invested in designing and implementing $\overline{\mathcal{A}}$.

Any $\overline{\text{IMP}}$ program s with sets of features $\mathbb{F}$ and valid configurations $\mathbb{K}$ is translated into a corresponding abstract program $\alpha(s)$ with corresponding set of features $\alpha(\mathbb{F})$ and set of valid configurations $\alpha(\mathbb{K})$. We define the translation recursively over the structure of α. The function α copies all basic statements of $\overline{IMP}$, and recursively calls itself for all sub-statements of compound statements other than `#if`. For example, $\alpha(\texttt{skip}) = \texttt{skip}$ and $\alpha(s_0\ ;\ s_1) = \alpha(s_0)\ ;\ \alpha(s_1)$. We discuss the rewrites for `#if` statements below.

In the rewrite, we associate a fresh feature name $Z \notin \mathbb{F}$, with every join abstraction $\boldsymbol{\alpha}^{\text{join}}$ (consequently written $\boldsymbol{\alpha}^{\text{join}}_{Z}$). The new feature Z is an abstract name (renaming) of the compound formula $\bigvee_{k\in\mathbb{K}} k$. It denotes the single valid configuration obtained from $\boldsymbol{\alpha}^{\text{join}}$. The new feature name is used to simplify conditions in the transformed code. The $\boldsymbol{\alpha}^{\text{join}}_{Z}$ rewrite is defined as follows:

$$
\boldsymbol{\alpha}^{\text{join}}_{Z}(\mathbb{F}) = \{Z\}, \quad \boldsymbol{\alpha}^{\text{join}}_{Z}(\mathbb{K}) = \{Z\}
$$

$$
\boldsymbol{\alpha}^{\text{join}}_{Z}(\texttt{\#if}\ (\theta)\ s) = \begin{cases} \texttt{\#if}\ (Z)\ \boldsymbol{\alpha}^{\text{join}}_{Z}(s) & \text{if } \bigvee_{k\in\mathbb{K}} k \models \theta \\ \texttt{\#if}\ (Z)\ \texttt{lub}(\boldsymbol{\alpha}^{\text{join}}_{Z}(s), \texttt{skip}) & \text{if sat}(\bigvee_{k\in\mathbb{K}} k \wedge \theta) \wedge \\ & \qquad \text{sat}(\bigvee_{k\in\mathbb{K}} k \wedge \neg\theta) \\ \texttt{\#if}\ (\neg Z)\ \boldsymbol{\alpha}^{\text{join}}_{Z}(s) & \text{if } \bigvee_{k\in\mathbb{K}} k \models \neg\theta \end{cases}
$$

In effect of applying the $\boldsymbol{\alpha}^{\text{join}}_{Z}$ transformation to any program s we obtain a single variant program, i.e. a SPL with only one valid product where the feature Z is enabled. It can be analyzed with existing single-program analyses. Note that it enables performing family-based analyses with implementations of single-program analyses, albeit with loss of precision. The newly introduced statement $\texttt{lub}(s_0, s_1)$ represents the least upper bound (join) of the results obtained

by executing s_0 and s_1. This is the only language-dependent aspect of `reconfigurator`. It can have different implementations depending on the programming language and the analysis we work with. In our case, we exploit the fact that $\overline{\mathcal{A}}[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!]$ ignores the branching condition (cf. Fig. 2) and use $\texttt{lub}(s_0, s_1) = \texttt{if } (n) \texttt{ then } s_0 \texttt{ else } s_1$ for some fixed integer n. Finally, observe that $\texttt{\#if } (\neg Z)\ \boldsymbol{\alpha}^{\text{join}}_{:Z}(s)$ is equivalent to `skip`, however it is useful to keep the statement in the program, which makes it easy to merge programs when we use compound abstractions (below).

The rewrite for projection only changes the set of legal configurations:

$$\boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\mathbb{F}) = \mathbb{F}, \quad \boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\mathbb{K}) = \{k \in \mathbb{K} \mid k \models \varphi\}, \quad \boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\texttt{\#if } (\theta)\ s) = \texttt{\#if } (\theta)\ \boldsymbol{\alpha}^{\text{proj}}_{\varphi}(s)$$

Note that the general scheme for the basic rewrites of `#if` statements can be summarized as $\alpha(\texttt{\#if } (\theta)\ s) = \texttt{\#if } (\overline{\alpha}(\theta))\ \overline{\alpha}(s, \theta)$, where $\overline{\alpha}$ are functions transforming the condition θ and the statement s. It is easy to extract $\overline{\alpha}(\theta)$ and $\overline{\alpha}(s, \theta)$ from the above rewrites for $\boldsymbol{\alpha}^{\text{join}}_{:Z}$ and $\boldsymbol{\alpha}^{\text{proj}}_{\varphi}$. We will use them in defining transformations for binary operators.

Now, for the case of parallel composition $\alpha_1 \otimes \alpha_2$, recall that the set $\alpha_1 \otimes \alpha_2(\mathbb{K})$ is the union of $\alpha_1(\mathbb{K})$ and $\alpha_2(\mathbb{K})$. However in the rewrite semantics, we are sometimes modifying the set of features. If $\alpha_1(\mathbb{F}) \neq \alpha_2(\mathbb{F})$ then some of valid configurations in $\alpha_1(\mathbb{K}) \cup \alpha_2(\mathbb{K})$ will not assign truth values to all features in $\alpha_1(\mathbb{F}) \cup \alpha_2(\mathbb{F})$. To take a meaningful union of configurations, we need to first unify their alphabets. To achieve this aim, each valid configuration can be extended by information that the missing features are excluded from it (negated). Now the rewrite rules for parallel composition are given by:

$$\alpha_1 \otimes \alpha_2(\mathbb{F}) = \alpha_1(\mathbb{F}) \cup \alpha_2(\mathbb{F})$$

$$\alpha_1 \otimes \alpha_2(\mathbb{K}) = \{k_1 \wedge \textstyle\bigwedge_{f \in \alpha_2(\mathbb{F}) \setminus \alpha_1(\mathbb{F})} \neg f \mid k_1 \in \alpha_1(\mathbb{K})\} \cup \{k_2 \wedge \textstyle\bigwedge_{f \in \alpha_1(\mathbb{F}) \setminus \alpha_2(\mathbb{F})} \neg f \mid k_2 \in \alpha_2(\mathbb{K})\}$$

$$\alpha_1 \otimes \alpha_2(\texttt{\#if } (\theta)\ s) = \begin{cases} \texttt{\#if } \left(\overline{\alpha_1}(\theta) \vee \overline{\alpha_2}(\theta)\right)\ \overline{\alpha_1}(s, \theta) & \text{if } \overline{\alpha_1}(s, \theta) = \overline{\alpha_2}(s, \theta) \\ \alpha_1(\texttt{\#if } (\theta)\ s); \alpha_2(\texttt{\#if } (\theta)\ s) & \text{otherwise} \end{cases}$$

Observe that the second case of the parallel composition transformation can only appear if the second case of a join transformation has been used somewhere in recursive rewriting of s (perhaps deep). All the other rewrites leave s intact. However, in such case the branches have disjoint feature alphabets, as every join is using a fresh feature name as parameter. This ensures that only one of the sequenced copies of s, $\overline{\alpha_1}(s, \theta)$ and $\overline{\alpha_2}(s, \theta)$, will actually be executed (and the other will amount to skip) in any given configuration of the product.

For sequential composition of abstractions $\alpha_2 \circ \alpha_1$ we use the following rewrites: $\alpha_2 \circ \alpha_1(\mathbb{F}) = \alpha_2(\alpha_1(\mathbb{F}))$, $\alpha_2 \circ \alpha_1(\mathbb{K}) = \alpha_2(\alpha_1(\mathbb{K}))$, and for the `#if` statement we have $\alpha_2 \circ \alpha_1(\texttt{\#if } (\theta)\ s) = \texttt{\#if } \left(\overline{\alpha_2}(\overline{\alpha_1}(\theta))\right)\ \overline{\alpha_2}(\overline{\alpha_1}(s, \theta), \overline{\alpha_1}(\theta))$.

▶ **Example 15.** Consider the program S_1': $\texttt{\#if}(A)\ \texttt{x} := \texttt{x} + 1; \texttt{\#if } (B)\ \texttt{x} := 1$ with $\mathbb{F} = \{A, B\}$, $\psi = A \vee B$, and $\mathbb{K}_\psi = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$. Then

$$\boldsymbol{\alpha}^{\text{join}}_{:Z} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}(S_1') = \texttt{\#if } (Z)\ \texttt{x} := \texttt{x} + 1; \texttt{\#if } (Z)\ \texttt{lub}(\texttt{x} := 1, \texttt{skip}) \tag{12}$$

The set of valid configurations after projection is changed to $\{A \wedge B, A \wedge \neg B\}$, and after join again to just $\{Z\}$. The obtained program has only one configuration, the one that satisfies Z. The projection does not change the statements of the program. The join rewrite however, simplifies the first `#if` (it is statically determined; cf. the first case of $\boldsymbol{\alpha}^{\text{join}}_{:Z}$ transformation), and joins the second statement with `skip` as it is unknown whether it will be executed or not, in the lack of information about the assignment to B in the

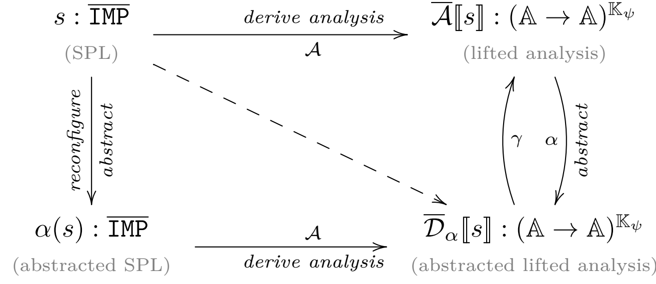


Figure 7 Illustration of *derive* vs *abstract*: $\overline{\mathcal{D}}_\alpha[\![s]\!] = \overline{\mathcal{A}}[\![\alpha(s)]\!]$.

abstracted program. Note that since Z is the only one valid configuration, the obtained program is equivalent to: $\texttt{x} := \texttt{x} + 1; \texttt{lub}(\texttt{x} := 1, \texttt{skip})$. Similarly, we can calculate: $\boldsymbol{\alpha}^{\text{join}}_{\text{`}Z\text{'}} \circ \boldsymbol{\alpha}^{\text{proj}}_{B}(S'_1) = \texttt{\#if}\ (Z)\ \texttt{lub}(\texttt{x} := \texttt{x} + 1, \texttt{skip}); \texttt{\#if}\ (Z)\ \texttt{x} := 1$.

Now consider $((\boldsymbol{\alpha}^{\text{join}}_{\text{`}Z\text{'}} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}) \otimes \boldsymbol{\alpha}^{\text{proj}}_{B})(S'_1)$. The new set of features is $\{Z, A, B\}$. The subset {A,B} is retained from the right projection component, and {Z} comes from the left join-project component. After extending the configurations of both components with negations of absent feature names we get the following set of valid configurations: $\mathbb{K}' = \{Z \land \neg A \land \neg B, \neg Z \land A \land B, \neg Z \land \neg A \land B\}$. The result of the left join-project operand is the program (12), and the right rewrite (projection) never changes the statements, so its result is identical to S'_1. Thus we are composing programs (12) and S'_1 using the parallel composition rewrites. Then $((\boldsymbol{\alpha}^{\text{proj}}_{Z} \circ \boldsymbol{\alpha}^{\text{join}}_{A}) \otimes \boldsymbol{\alpha}^{\text{proj}}_{B})(S'_1)$ is:

$$\texttt{\#if}\ (Z \lor A)\ \texttt{x} := \texttt{x} + 1; \texttt{\#if}\ (Z)\ \texttt{lub}(\texttt{x} := 1, \texttt{skip}); \texttt{\#if}\ (B)\ \texttt{x} := 1$$

The first `#if` has been unified using the first case of the transformation for $\otimes$, and the second `#if` is transformed into two copies of the statement with different guards, using the second case of the rewrite definition. For any legal configuration in $\mathbb{K}'$ at most one of them does not reduce to `skip`. ◀

Now the analysis $\overline{\mathcal{A}}[\![\alpha(s)]\!]$ and $\overline{\mathcal{D}}_\alpha[\![s]\!]$ coincide up to renaming of valid configurations. So the `reconfigurator` together with an existing implementation of $\overline{\mathcal{A}}$ gives us the abstracted analysis $\overline{\mathcal{D}}_\alpha$. The above equality is illustrated by Fig. 1.

▶ **Theorem 16.**
$\forall s \in \mathit{Stm}, \alpha : \mathbb{A}^{\mathbb{K}_\psi} \to \mathbb{A}^{\alpha(\mathbb{K}_\psi)} \in \mathit{Abs}, \overline{d} \in \mathbb{A}^{\alpha(\mathbb{K}_\psi)} :\ \overline{\mathcal{D}}_\alpha[\![s]\!]\,\overline{d} = \overline{\mathcal{A}}[\![\alpha(s)]\!]\,\overline{d}$ [5].

▶ **Example 17.** Consider the program S_1 from Example 1 with $\mathbb{K}_\psi = \{A \land B, A \land \neg B, \neg A \land B\}$. We have calculated in Example 13 that $\overline{\mathcal{D}}_{\boldsymbol{\alpha}^{\text{join}}_{A}}[\![S_1]\!]([\texttt{x} \mapsto \top]) = ([\texttt{x} \mapsto 1])$. We now calculate $\overline{\mathcal{A}}[\![\boldsymbol{\alpha}^{\text{join}}_{A,\text{`}Z\text{'}}(S_1)]\!]([\texttt{x} \mapsto \top])$ (here $\boldsymbol{\alpha}^{\text{join}}_{A,\text{`}Z\text{'}} = \boldsymbol{\alpha}^{\text{join}}_{\text{`}Z\text{'}} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}$):

$$([\texttt{x} \mapsto \top]) \overset{\overline{\mathcal{A}}[\![\texttt{x}:=0]\!]}{\longmapsto} ([\texttt{x} \mapsto 0]) \overset{\overline{\mathcal{A}}[\![\texttt{\#if}\ (Z)\ \texttt{x}:=\texttt{x}+1]\!]}{\longmapsto} ([\texttt{x} \mapsto 1]) \overset{\overline{\mathcal{A}}[\![\texttt{\#if}\ (Z)\ \texttt{lub}(\texttt{x}:=1,\texttt{skip})]\!]}{\longmapsto} ([\texttt{x} \mapsto 1]) \quad ◀$$

6 Evaluation

Recall that there are two ways to speed up lifted analyses: improving *representation* and increasing *abstraction*. First, we will compare the performance of the two using an unoptimized

[5] The proof of this theorem is in [14, App. F].

Benchmark	avg. $\lvert\mathbb{K}_\psi\rvert$	$\lvert\mathbb{F}\rvert$	LOC	max variab. mth	$\lvert\mathbb{K}_\psi\rvert$	$\lvert\mathbb{F}\rvert$	LOC
`Prevayler`	N=1.3	5	8,000	`P'F'.publisher()`	N=8	3	10
`BerkelyDB`	N=1.6	42	84,000	`DBRunAct.main()`	N=40	7	165
`GPL`	N=3.9	18	1,350	`Vertex.display()`	N=106	9	31

Figure 8 Characteristics of our three SPL benchmarks (average #configurations in all methods in SPL, total #features, and LOC) along with, for each SPL, its method with maximum variability (#configurations, local #features, and LOC).

lifted analysis as a baseline. Then, we demonstrate that abstraction may be used to turn previously infeasible analysis into feasible ones. Finally, we consider example scenarios that use projection and join and show that abstraction may be applied to an entire product line or when just analyzing a single method.

For our experiments, we use an existing implementation of lifted data-flow analyses for Java Object-Oriented SPLs [5]. The implementation is based on SOOT's intra-procedural data-flow analysis framework [24] for analyzing Java programs. It uses CIDE (Colored IDE) [16] to annotate statements using background colors rather than `#ifdef` directives. Every feature is thus associated with a unique color.

We will consider an unoptimized lifted intra-procedural analysis, known as $\mathcal{A}2$ (from [5]), that uses $|\mathbb{K}_\psi|$-tuples of analysis information, one analysis value per configuration. Also, we consider $\mathcal{A}3$ (from [5]) which is the same analysis as $\mathcal{A}2$, but with improved representation via sharing of analysis-equivalent configurations using a high-performance bit vector library [22]. So $\mathcal{A}3$ is an optimized version of $\mathcal{A}2$ where shared representation is used for representing sets of configurations (i.e. components of tuples) with equivalent analysis information. Note that $\mathcal{A}2$ corresponds to $\overline{\mathcal{A}}$ in Fig. 2 and we will thus refer to it as $\overline{\mathcal{A}}$, while we will use $\overline{\mathcal{S}}$ for the analysis with sharing ($\mathcal{A}3$ in [5]). The performance of abstracted analyses depends on the size of tuples they work on. Therefore as variability abstractions, we have chosen $\overline{\mathcal{D}}_{\boldsymbol{\alpha}^{\text{join}}}$ which joins together (confounds) information from all configurations down to just one abstracted analysis value, and $\overline{\mathcal{D}}_{\boldsymbol{\alpha}^{\text{proj}}_{N/2}\otimes\boldsymbol{\alpha}^{\text{join}}_{N/2}}$ (where $N = |\mathbb{K}_\psi|$) which is a parallel composition of a projection of 1/2 (randomly selected) configurations and a *join* of the remaining 1/2 configurations. We abbreviate them as $\overline{\mathcal{D}}_1$ and $\overline{\mathcal{D}}_{N/2}$ in the following. We have chosen those variability abstractions because they represent the coarsest abstraction $\overline{\mathcal{D}}_1$ that works on 1-sized tuples, and the medium abstraction $\overline{\mathcal{D}}_{N/2}$ that works on $N/2$-sized tuples. Any other abstraction will have a speed up anywhere between $\overline{\mathcal{A}}$ (no abstraction), $\overline{\mathcal{D}}_{N/2}$ (medium abstraction) and $\overline{\mathcal{D}}_1$ (maximum abstraction). It thus quantifies the potential of abstractions.

For our experiment, we have chosen two analyses: *reaching definitions* and *uninitialized variables*, for which we derived the corresponding definitions of abstracted lifted analysis. We use three SPL benchmarks [16]: Graph PL (GPL) is a small desktop application with intensive feature usage; Prevayler is a slightly larger product line with low feature usage; and BerkelyDB is a larger database library with moderate feature usage. Fig. 8 summarises relevant characteristics for each benchmark: the average number of valid configurations in all methods in the SPL, the total number of features in the entire SPL, the total number of lines of code (LOC). Also, for each SPL, the figure details information about the method with the highest variability (most configurations): its number of valid configurations, features, and lines of code.

Performance. Fig. 9 shows the time it takes to run each of our three maximum variability methods, as a relative comparion between $\overline{\mathcal{A}}$ (baseline) and $\overline{\mathcal{S}}$ (sharing) *vs* $\overline{\mathcal{D}}_{N/2}$ (medium

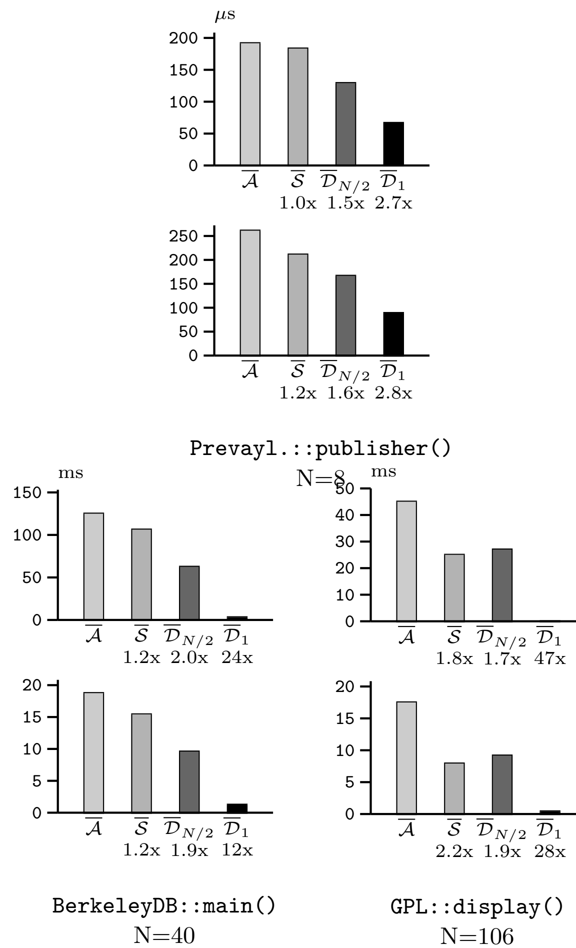


Figure 9 Analysis time for *reaching definitions* (above) and *uninitialized variables* (below): $\overline{\mathcal{A}}$ (baseline) and $\overline{\mathcal{S}}$ (sharing) vs. $\overline{\mathcal{D}}_{N/2}$ (medium abstraction) and $\overline{\mathcal{D}}_1$ (maximum abstraction).

abstraction) and $\overline{\mathcal{D}}_1$ (maximum abstraction). The experiments are executed on a 64-bit Intel®Core^{TM} i5 CPU with 8 GB memory. All times are reported as averages over ten runs with the highest and lowest number removed. For each benchmark method, we give the speed up factor relative to the baseline (normalized with factor 1) and recall the number of configurations, N.

Our experiment confirms previous results that sharing is indeed effective and especially so for larger values of N [5]. On our methods, it translates to speed ups (i.e., $\overline{\mathcal{A}}$ *vs* $\overline{\mathcal{S}}$) anywhere between 3% faster (for N=8) and slightly more than twice as fast (for N=106). We also observe that abstraction is not surprisingly significantly faster than unabstracted analyses (i.e., $\overline{\mathcal{D}}$ *vs* $\overline{\mathcal{A}}$ and $\overline{\mathcal{S}}$); i.e., abstraction yields significant performance gains, especially for benchmarks with higher variability. For `GPL` with N=106, we see a dramatic 47 and 28 times speed up depending on the analysis (i.e., $\overline{\mathcal{D}}_1$ *vs* $\overline{\mathcal{A}}$). Also, we note that increased abstraction is up to 26 times faster than improved representation (i.e., $\overline{\mathcal{D}}_1$ *vs* $\overline{\mathcal{S}}$). In general, it is obviously possible to combine the benefits from representation and abstraction to yield even more efficient analyses.

From Infeasible to Feasible Analysis. Of course, for very large values of N, analyses may become impractically slow or infeasible. As an experiment, we took a large method

```
void   main(..)  {
1        ..  int doAction = 0; ..
2        #ifdef Cleaner
3        if (..)  doAction = CLEAN;
4        #endif
5        #ifdef INCompresser
6        if (..)  doAction = COMPRESS;
7        #endif
8        if (..)  doAction = CHECKPOINT;
9        #ifdef Statistics
10       if (..)  doAction = DBSTATS;
11       #endif
12       ..  switch (doAction) { ..  } ..
}
```

Figure 10 Code fragment extracted from `BerkeleyDB::main()` with N=40.

(`processFile()` from `BerkeleyDB`) and kept adding unconstrained variability manually. For $N=2^{13}$=8,192 configurations, the analysis $\overline{\mathcal{A}}$ took 138 seconds. For $N=2^{14}$=16,384, it ran more than ten minutes until it eventually produced an out-of-memory error. In contrast, variability abstraction $\overline{\mathcal{D}}_1$ analyses the same high variability method in less than 8 ms (albeit less precisely). Hence, abstraction can not only speed up analyses, but also turn previously infeasible analyses feasible.

Projection on Entire SPL. GPL is a family of classical graph applications with variability on its representation and algorithms. For instance, the features `Directed` and `Undirected` control whether or not graphs are *directed*; `Weighted` and `Unweighted` control whether or not the graphs are *weighted*; and, the features `BFS` and `DFS` control the search algorithm used (*breadth-first search* or *depth-first search*). It is common industrial practice, to ship products with a subset of configurations, and thereby functionality. Here, we may use projection to *disable* features `BFS` and `Undirected`, along with any features that only work on undirected graphs: (`Connected`, `MSTKruskal`, and `MSTPrim` for implementing *connected components* and *minimum spanning trees* algorithms) which can be obtained from GPL's feature model, detailing such feature dependencies. With this projection (abstraction), the configuration space of GPL is reduced from 528 to 370 valid configurations. This, in turn, cuts analysis time of reaching definitions in half (from 90ms to 49ms). For 123 out of 135 methods, the abstracted analysis computes the exact same analysis information. For larger product lines and projections, lots of time may be saved in this way.

Join on One Method. Figure 10 shows a fragment extracted from `BerkeleyDB`'s `main()` method with N=40 valid configurations. A local variable, `doAction` is defined and initialized to zero, after which it is conditionally assigned three times in statements guarded by `#ifdef`s. (Actually, there are two more similar `#ifdef`s involving features `Evictor` and `DeleteOp`, but we have omitted those for brevity in the code fragment.) We can use a join abstraction of the reaching definitions analysis to compute what are the possible values (definitions) that *reach* the condition of the `switch` statement in line 12. An abstracted analysis would be able to determine that these are the assignments in lines 1, 3, 6, 8, and 10, by analyzing only *one* crudely over-approximated configuration instead of all (N=40) configurations. In general, by inspecting the structure of the code and the features used, we can tailor abstactions that can analyze individual methods much faster than analyzing all configurations.

7 Related Work

Static analyses can be accelerated by devicing more efficient representations or by introducing abstraction. In family-based analysis for software product lines the representation improvements primarily rely on sharing state information for variants with analysis-equivalent information (which implies reducing redundant computation). This can optimize the analyses considerably [5, 6, 18]. However, in the worst case, the number of variants that a lifted analysis has to consider is still inherently exponential in the number of features, $|\mathbb{F}|$. Thus with a large number of features lifted analyses may become impractical or even infeasible. In this work we have taken the alternate route of using abstraction. Our experiments show that abstraction introduces speed-ups independently of representation gains. Thus our results can be beneficially combined with efficient representations.

An efficient implementation of lifted analysis formulated within the IFDS framework [21] for inter-procedural distributive environments was proposed in SPL$^{\text{LIFT}}$ [4]. It uses binary decision diagrams to represent shared feature constraints. The authors have found that the running time of analysing all variants in a family is close to the analysis of a single-program. In such case, further benefit of applying abstraction, as presented in this paper, is unlikely to bring any significant improvement. However, notice that the method of SPL$^{\text{LIFT}}$ is limited only to distributive data-flow analysis encoded within the IFDS framework. Many analyses, including constant propagation, are not distributive and hence cannot be expressed in IFDS. Let alone static analyses that are not expressible as data-flow analyses (including type checking, model-checking, etc).

The formal developments in this paper are based on *variational abstract interpretation*, a formal methodology for systematic derivation of lifted analyses for `#ifdef`-based product lines, proposed in [19]. The method is based on the calculational approach to abstract interpretation of Cousot [9], applied and contextualized to product lines. In that work [19], calculations are used to derive a directly operational *lifted analysis* which is *correct* by construction. In the present paper, we assume that lifted analyses exist (possibly obtained using the methodology of [19]), and focus on abstracting variability using them. We devise an expressive calculus for specifying abstraction operators. Also, all abstractions specifiable in our calculus are now automatically executable. Implementing abstractions as program transformations looks similar to the framework defined in [13] for designing source-to-source program transformations by abstract interpretation of program semantics.

A good collection of analyses that have been lifted manually is presented in the survey [23]. We should remark, that the join operation $\boldsymbol{\alpha}^{\text{join}}$ allows applying single program analyses to program families, even if with precision loss. In that sense, the our approach is the first ever method that can *automatically* lift single program analyses to work on program families. Besides the family-based strategy, the survey [23] identifies a *sampling strategy* as a suitable way of analyzing product lines (see also [1]). In the sampling strategy only a random subset of products is analyzed. We remark that once the sample is selected, our projection operator $\boldsymbol{\alpha}^{\text{fproj}}_{\varphi}$ can be used to realize the sampling strategy in a simultanous way by exploiting an existing family-based analysis.

In fact, the abstraction specification framework of Section 3 allows specifying any analysis in the spectrum between a fully family-based analyses, and a single variant, *single product-based*, analysis. We can specify abstractions that select (sample) any subsets of configurations and then analyze this subset with selected choice of precision, either all variants precisely, like in sampling, or confounding some executions for efficiency. In this sense, we show how to design analyses placed anywhere in the design spectrum painted in [23]. Consider, the *feature-*

based analysis strategy as an example. In this strategy an analysis explores the program code feature-by-feature (as opposed to configuration-by-configuration). Analyses following this strategy can now be systematically obtained using our abstractions, by projecting away (ignoring) all but one feature and running a single program analysis on the result. This is quite remarkable. It has been well recognized that designing such analyses is very difficult, yet now there exists a systematic way of doing that, so it is no longer an impenetrable art.

8 Conclusion

We have defined variability-aware abstractions given as Galois connections, and used them to derive efficient and correct-by-construction abstract analyses of program families. We have designed a calculus for the abstractions, and shown how abstractions specified in this language can be applied not only on analyses, but also on programs, obtaining a convenient implementation strategy of the abstractions in form of a source-to-source `reconfigurator` transformation.

We have proved the main results (Theorem 11 and Theorem 16) for constant propagation analysis and extracted a general proof methodology that holds for any other monotone and computable analysis that can be lifted. We have derived the abstracted definition of `#if` with the lowest precision. Improvements of the precision are possible once the analysis is known.

The `reconfigurator` transformation presently requires that the programming language is able to express *sequential composition* (e.g., "`;`" in `IMP`) and *join of statements* (i.e., `lub` as in "$\sqcup$") with respect to the analysis in question. It would be interesting to consider lifting those assumptions in future, and apply this method to more modeling and programming languages.

We evaluated the method on three Java-based product lines. We found that the abstractions improve performance of analyses independently of improvements in the data representations used in the implementations of these analyses. This indicates that the proposed abstraction strategies will be instrumental in tackling error finding analysis in large configurable software systems, like the Linux kernel. Indeed we have developed these techniques with the intention of scaling error finding tools to such challenging cases in future. Besides this, we would like to experiment with applying these abstraction techniques to alternative quality assurance methods including model checking, and testing.

References

1. Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *35th International Conference on Software Engineering, ICSE'13*, pages 482–491, 2013.
2. Don Batory. Feature models, grammars, and propositional formulas. In *9th Int'l Software Product Lines Conf., SPLC'05*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.
3. Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wasowski. Three cases of feature-based variability modeling in industry. In *17th Int'l Conf. Model-Driven Engineering Languages and Systems, MODELS'14*, pages 302–319, 2014.
4. Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spl^{lift}: Statically analyzing software product lines in minutes instead of years. In *ACM SIGPLAN Conference on PLDI'13*, pages 355–364, 2013.

5 Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. *Transactions on Aspect-Oriented Software Development*, 10:73–108, 2013.

6 Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *33th International Conference on Software Engineering, ICSE'11*, pages 321–330, 2011.

7 Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

8 Patrick Cousot. Types as abstract interpretations. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'97*, pages 316–331, 1997.

9 Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

10 Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *6th Annual ACM Symposium on Principles of Programming Languages, POPL'79*, pages 269–282, 1979.

11 Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2–3):103–179, 1992.

12 Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Autom. Softw. Eng.*, 6(1):69–95, 1999.

13 Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'02*, pages 178–190, 2002.

14 Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Variability abstractions: Trading precision for speed in family-based analyses (extended version). *CoRR*, abs/1503.04608, 2015.

15 Eric Goubault, Dominique Guilbaud, Anne Pacalet, Basile Starynkevitch, and Franck Védrine. A simple abstract interpreter for threat detection and test case generation. In *WAPATV'01, with ICSE'01*, Toronto, 2001.

16 Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, Germany, May 2010.

17 Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *30th International Conference on Software Engineering, ICSE'08*, pages 311–320, Leipzig, Germany, 2008. ACM.

18 Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3):14, 2012.

19 Jan Midtgaard, Claus Brabrand, and Andrzej Wasowski. Systematic derivation of static analyses for software product lines. In *13th International Conference on Modularity, MODULARITY'14, 2014*, pages 181–192, 2014.

20 Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, Secaucus, USA, 1999.

21 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95*, POPL'95, pages 49–61, 1995.

22 The colt project: Open source libraries for high performance scientific and technical computing in java. CERN: European Organization for Nuclear Research.

23 Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6, 2014.

24 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot – a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research*, page 13, 1999.

25 Glynn Winskel. *The Formal Semantics of Programming Languages.* Foundation of Computing Series. The MIT Press, 1993.

Optimization Coaching for JavaScript

Vincent St-Amour[1] **and Shu-yu Guo**[2]

1 **PLT @ Northeastern University**
Boston, Massachusetts, USA
`stamourv@ccs.neu.edu`

2 **Mozilla Research**
San Francisco, California, USA
`shu@mozilla.com`

Abstract

The performance of dynamic object-oriented programming languages such as JavaScript depends heavily on highly optimizing just-in-time compilers. Such compilers, like all compilers, can silently fall back to generating conservative, low-performance code during optimization. As a result, programmers may inadvertently cause performance issues on users' systems by making seemingly inoffensive changes to programs. This paper shows how to solve the problem of silent optimization failures. It specifically explains how to create a so-called *optimization coach* for an object-oriented just-in-time-compiled programming language. The development and evaluation build on the SpiderMonkey JavaScript engine, but the results should generalize to a variety of similar platforms.

1998 ACM Subject Classification D.2.3 [Software Engineering] Coding Tools and Techniques, D.3.4 [Programming Languages] Processors – Compilers

Keywords and phrases Optimization Coaching, JavaScript, Performance Tools

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.271

Supplementary Material ECOOP Artifact Evaluation approved artifact available at
`http://dx.doi.org/10.4230/DARTS.1.1.5`

1 Optimization Coaching for the Modern World

An optimization coach [22] opens a dialog between optimizing compilers and programmers. It thus allows the latter to take full advantage of the optimization process. Specifically, coaches provide programmers with actionable recommendations of changes to their programs to trigger additional optimizations. Notably, the changes may not preserve the semantics of the program.

Our experiences with a prototype optimization coach for Racket show promising results. This prototype exploits Racket's [9] simple ahead-of-time byte-compiler,[1] which performs basic optimizations. The general idea of optimization coaching ought to apply beyond languages with functional cores and simple compilers.

Unsurprisingly, scaling coaching to object-oriented languages with advanced compilers presents challenges. An object-oriented programming style gives rise to *non-local optimization failures*, that is, the compiler may fail to optimize an operation in one part of the program because of properties of a different part of the program. Advanced just-in-time (JIT) compilers

[1] Racket also includes a just-in-time code generator that does not perform many optimizations.

© Vincent St-Amour and Shu-yu Guo;
licensed under Creative Commons License CC-BY

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 271–295

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

introduce a *temporal dimension* to the compilation and optimization process, that is, the compiler may compile the same piece of code multiple times, potentially performing different optimizations each time. Advanced compilers may also apply *optimization tactics* when optimizing programs, that is, they use batteries of related and complementary optimizations when compiling some operations.

This paper presents new ideas on optimization coaching that allow it to scale to dynamic object-oriented languages with state-of-the-art JIT compilers. Our prototype optimization coach works with the SpiderMonkey[2] JavaScript [7] engine, which is included in the Firefox[3] web browser.

In this paper, we

- describe optimization coaching techniques designed for object-oriented languages with state-of-the-art compilers
- present an evaluation of the recommendations provided by our optimization coach for SpiderMonkey.

The rest of the paper is organized as follows. Sections 2 and 3 provide background on optimization coaching and on the SpiderMonkey JavaScript engine. Section 4 describes the optimizations that our prototype supports. Section 5 sketches out its architecture. Section 6 outlines the challenges of coaching in an object-oriented setting and describes our solutions, and Section 7 does likewise for the challenges posed by JIT compilation and optimization tactics. Section 8 presents coaching techniques that ultimately were unsuccessful. We then present evaluation results in Section 9, compare our approach to related work and conclude.

Prototype. Our prototype optimization coach is available in source form.[4] It depends on an instrumented version of SpiderMonkey whose source is also available.[5]

2 Background: Optimization Coaching

Because modern programming languages heavily rely on compiler optimizations for performance, failure to apply certain key optimizations is often the source of performance issues. To diagnose these performance issues, programmers need insight about what happens during the optimization process.

This section first discusses an instance of an optimization failure causing a hard-to-diagnose performance issue. The rest of the section then provides background on how optimization coaching provides assistance in these situations, and introduces some key technical concepts from previous work on coaching.

2.1 A Tale from the Trenches

The Shumway project,[6] an open-source implementation of Adobe Flash in JavaScript, provides an implementation of ActionScript's parametrically polymorphic `Vector` API,[7] which includes a `forEach` method. This method takes a unary kernel function `f` as its argument and calls it

[2] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey
[3] https://www.mozilla.org/en-US/firefox/
[4] https://github.com/stamourv/jit-coach
[5] https://github.com/stamourv/gecko-dev/tree/profiler-opt-info
[6] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Shumway
[7] http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/Vector.html

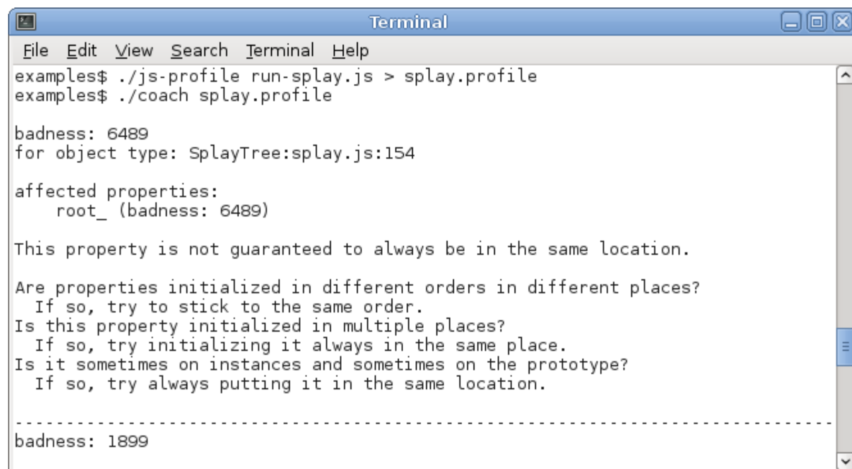


Figure 1 Excerpt from the coaching report for a splay tree implementation.

once for each element in the vector, passing that element as the argument to `f`. Initially, the Shumway implementors wrote a single implementation of the `forEach` method and used it for all typed variants of `Vector`.

This initial implementation performed poorly. Unbeknownst to Shumway's implementors, this implementation strategy triggers optimization failures inside JavaScript engines. If the compiler observes code to be polymorphic, it may not apply crucial optimizations. For instance, the compiler may be unable to determine a monomorphic context for the element accesses and the calling of the kernel function, prohibiting optimizations such as inlining.

Eventually, the Shumway engineers reverse engineered the JIT's opaque optimization decisions and could then diagnose the problem. They determined that performance could be recouped by cloning `forEach`'s implementation for variants that needed high performance (e.g., vectors of integers), as the JIT would then observe monomorphic accesses and call sites. While the compiler lacked the necessary context to make the appropriate tradeoff decision, the Shumway engineers were able to, once they understood the optimization decisions.

2.2 Optimization Coaching in a Nutshell

Failures such as those experienced by the first Shumway implementation are hard to diagnose and solve for two main reasons. First, optimizers fail silently; programmers are never informed that an optimization failed. Second, getting to the root causes of these failures requires skills and knowledge that are out of reach for most programmers. Those skills include auditing the compiler's output, reverse engineering the optimizer's decisions, etc.

Optimization coaches help programmers get more out of their optimizers without requiring such knowledge and with a minimum of effort. They achieve this feat by reporting *optimization near misses*. Near misses are optimizations that the compiler did not apply to their program – either due to a lack of information, or because doing so may be unsound in some cases – but could apply safely if the source program were changed in a certain way.

For example, consider the excerpt from a splay tree implementation in Figure 2. The `isEmpty` method may find the `root_` property either on `SplayTree` instances (if the `insert` method has been called) or on the `SplayTree` prototype (otherwise). Hence, the JavaScript engine cannot specialize the property access to either of these cases and instead generates code that can handle both of them. The generated code is thus much slower than necessary.

```
// constructor
function SplayTree() {};
// default value on the prototype
SplayTree.prototype.root_ = null;

SplayTree.prototype.insert = function(key, value) {
  // regular value on instances
  ... this.root_ = new SplayTree.Node(key, value); ...
};

SplayTree.prototype.isEmpty = function() {
  // property may be either on the instance or on the prototype
  return !this.root_;
};
```

Figure 2 Splay tree implementation with an optimization near miss.

```
function SplayTree() {
    // default value on instances
    this.root_ = null;
};
```

Figure 3 Improved splay tree constructor, without near miss.

Coaches supplement near miss reports with concrete recommendations of program changes that programmers can apply. These modifications may make the compiler's analysis easier or may rule out corner cases, with an end result of the compiler succeeding to apply previously missed optimizations. Figure 1 shows the coach's diagnosis and recommendations of program changes that may resolve the near miss.

These recommendations are not required to preserve programs' exact semantics. In other words, coaches may recommend changes that would be beyond the reach of optimizing compilers, which are limited to semantics-preserving transformations. Programmers remain in control and are free to veto any recommendation that would lead to semantic, or structural, changes that they deem unreasonable.

In our splay tree example, the compiler cannot move `root_`'s default value to instances; this would change the behavior of programs that depend on the property being on the prototype. Programmers, on the other hand, are free to do so and may rewrite the program to the version from Figure 3, which consistently stores the property on instances, and does not suffer from the previous near miss.

2.3 Optimization Coaching Concepts

To provide the necessary background to describe this paper's technical contributions, we now provide an overview of existing optimization coaching concepts.

At a high level, an optimization coach operates in four phases.

- First, instrumentation code inside the optimizer logs optimization decisions during compilation. This instrumentation distinguishes between *optimization successes*, i.e., optimizations that the compiler applies to the program, and *optimization failures*, i.e., optimizations that it cannot apply. These logs include enough information to reconstruct the optimizer's reasoning post facto. Section 4 describes the information recorded by our prototype's instrumentation, and Section 7.1 explains our approach to instrumentation in a JIT context.
- Second, after compilation, an offline analysis processes these logs. The analysis phase is responsible for producing high-level, human-digestible near miss reports from the low-level optimization failure events recorded in the logs. It uses a combination of

optimization-agnostic techniques and optimization-specific heuristics. We describe some of these techniques below.

- Third, from the near miss reports, the coach generates recommendations of program changes that are likely to turn these near misses into optimization successes. These recommendations are generated from the causes of individual failures as determined during compilation and from metrics computed during the analysis phase.
- Finally, the coach shows reports and recommendations to programmers. The interface should leverage optimization analysis metrics to visualize the coach's rankings of near misses or display high-estimated-impact recommendations only.

To avoid overwhelming programmers with large numbers of low-level reports, an optimization coach must carefully curate and summarize its output. In particular, it must restrict its recommendations to those that are both likely to enable further optimizations and likely to be accepted by the programmer. A coach uses three main classes of techniques for that purpose: pruning, ranking and merging.

Pruning. Not all optimization failures are equally interesting to programmers. For example, showing failures that do not come with an obvious source-level solution, or failures that are likely due to intentional design choices, would be a waste of programmer time. Coaches therefore use heuristics that decide to remove optimization failures from the coach's reports. Optimization failures that remain after pruning constitute near misses, and are further refined via merging.

Our previous work describes several pruning techniques, such as *irrelevant failure pruning*, which we discuss in Section 7.2.1. Section 7.1.2 introduces a new form of pruning based on profiling information.

Ranking. Some optimization failures have a larger impact on program performance than others. A coach must rank its reports based on their expected performance impact to allow programmers to prioritize their responses. In order to do so, the coach computes a *badness* metric for each near miss, which estimates its impact on performance.

Our previous work introduces static heuristics to compute badness. Section 7.1.2 introduces the new dynamic heuristic that our prototype uses.

Merging. To provide a high-level summary of optimization issues affecting a program, a coach should consolidate sets of related reports into single summary reports. Different merging techniques use different notions of relatedness.

These summary reports have a higher density of information than individual near miss reports because they avoid repeating common information, which may include cause of failure, solution, etc. depending on the notion of relatedness. They are also more efficient in terms of programmer time. For example, merging reports with similar solutions or the same program location, allows programmers to solve multiple issues at the same time.

When merging reports, a coach must respect preservation of badness which, for summary reports, is the sum of that of the merged reports. The sum of their expected performance impacts is a good estimation of the estimated impact of the summary report. The increased badness value of summary reports causes them to rank higher than their constituents would separately. Because these reports have a higher impact-to-effort ratio, having them high in the rankings increases the actionability of the tool's output.

Our previous work introduces two merging techniques: *causality merging* and *locality merging*. This work introduces three additional kinds of merging, *by-solution merging* (Section 6.3), *by-constructor merging* (Section 6.4) and *temporal merging* (Section 7.1.3).

3 Background: The SpiderMonkey JavaScript Engine

This section surveys the aspects of SpiderMonkey that are relevant to this work.

3.1 Compiler Architecture

Like other modern JavaScript engines,[8,9,10] SpiderMonkey is a multi-tiered engine that uses type inference [13], type feedback [1], and optimizing just-in-time compilation [2] based on the SSA form [5], a formula proven to be well suited for JavaScript's dynamic nature. Specifically, it has three tiers: the interpreter, the baseline JIT compiler, and the IonMonkey (Ion) optimizing JIT compiler.

In the interpreter, methods are executed without being compiled to native code or optimized. Upon reaching a certain number of executions,[11] the baseline JIT compiles methods to native code. Once methods become hotter still and reach a second threshold,[12] Ion compiles them. The engine's gambit is that most methods are short-lived and relatively cold, especially for web workloads. By reserving heavyweight optimization for the hottest methods, it strikes a balance between responsiveness and performance.

3.2 Optimizations in Ion

Optimizations in Ion

Because Ion performs the vast majority of SpiderMonkey's optimizations, our work focuses on coaching those. Ion is an optimistic optimizing compiler, meaning it assumes types and other observed information gathered during baseline execution to hold for future executions, and it uses these assumptions to drive the optimization process.

Types and layout. For optimization, the information SpiderMonkey observes mostly revolves around type profiling and object layout inference. In cases where inferring types would require a heavyweight analysis, such as heap accesses and function calls, SpiderMonkey uses type profiling instead. During execution, baseline-generated code stores the result types for heap acccesses and function calls for consumption by Ion.

At the same time, the runtime system also gathers information to infer the layouts of objects, i.e., mappings of property names to offsets inside objects. These layouts are referred to as "hidden classes" in the literature. This information enables Ion to generate code for property accesses on objects with known layout as simple memory loads instead of hash table lookups.

The applicability of Ion's optimizations is thus limited by the information it observes. The observed information is also used to seed a number of time-efficient static analyses, such as intra-function type inference.

---

[8] `https://developers.google.com/v8/intro`
[9] `http://www.webkit.org/projects/javascript/`
[10] `http://msdn.microsoft.com/en-us/library/aa902517.aspx`
[11] At the time of this writing, 10.
[12] At the time of this writing, 1000.

Bailouts. To guard against changes in the observed profile information, Ion inserts dynamic checks [15]. For instance, if a single callee is observed at a call site, Ion may optimistically inline that callee, while inserting a check to ensure that no mutation changes the binding referencing the inlinee. Should such a dynamic check fail, execution aborts from Ion-generated code and resumes in the non-optimized – and therefore safe – code generated by the baseline JIT compiler.

Optimization tactics. As a highly optimizing compiler, Ion has a large repertoire of optimizations at its disposal when compiling key operations, such as property accesses. These optimizations are organized into *optimization tactics*. When compiling an operation, the compiler attempts each known optimization strategy for that kind of operation in order – from most to least profitable – until one of them applies.

A tactic's first few strategies are typically highly specialized optimizations that generate extremely efficient code, but apply only in limited circumstances, e.g., accessing a property of a known constant object. As compilation gets further into a tactic, strategies become more and more general and less and less efficient, e.g., polymorphic inline caches, until it reaches fallback strategies that can handle any possible situation but carry a significant performance cost, e.g., calling into the VM.

4 Optimization Corpus

Conventional wisdom among JavaScript compiler engineers points to property and element accesses as the most important operations to be optimized. For this reason, our prototype focuses on these two classes of operations.

For both, the instrumentation code records similar kinds of information. The information uniquely identifies each operation affected by optimization decisions, i.e., source location, type of operation and parameters. Additionally, it records information necessary to reconstruct optimization decisions themselves, i.e., the sets of inferred types for each operand, the sequence of optimization strategies attempted, which attempts were successful, which were not and why. This information is then used by the optimization analysis phase to produce and process near miss reports.

The rest of this section describes the relevant optimizations with an eye towards optimization coaching.

4.1 Property Access and Assignment

Conceptually, JavaScript objects are open-ended maps from strings to values. In the most general case, access to an object property is at best a hash table lookup, which, despite being amortized constant time, is too slow in practice. Ion therefore applies optimization tactics when compiling these operations so that it can optimize cases that do not require the full generality of maps. We describe some of the most important options below.

Definite slot. Consider a property access `o.x`. In the best case, the engine observes `o` to be monomorphic and with a fixed layout. Ion then emits a simple memory load or store for the slot where `x` is stored. This optimization's prerequisites are quite restrictive. Not only must all objects that flow into `o` come from the same constructor, they must also share the same fixed layout. An object's layout is easily perturbed, however, for example by adding properties in different orders.

Polymorphic inline cache. Failing that, if multiple types of plain JavaScript objects[13] are observed to flow to `o`, Ion can emit a polymorphic inline cache (PIC) [14]. The PIC is a self-patching structure in JIT code that dispatches on the type and layout of `o`. Initially, the PIC is empty. Each time a new type and layout of `o` flows into the PIC during execution, an optimized *stub* is generated that inlines the logic needed to access the property `x` for that particular layout of `o`. PICs embody the just-in-time philosophy of not paying for any expensive operation ahead of time. This optimization's prerequisites are less restrictive than that of definite slots, and it applies for the majority of property accesses that do not interact with the DOM.

VM call. In the worst case, if `o`'s type is unknown to the compiler, either because the operation is in cold code and has no profiling information, or because `o` is observed to be an exotic object, then Ion can emit only a slow path call to a general-purpose runtime function to access the property.

Such slow paths are algorithmically expensive because they must be able to deal with any aberration: `o` may be of a primitive type, in which case execution must throw an error; `x` may be loaded or stored via a native DOM accessor somewhere on `o`'s prototype chain; `o` may be from an embedded frame within the web page and require a security check; etc. Furthermore, execution must leave JIT code and return to the C++ VM. Emitting a VM call is a last resort; it succeeds unconditionally, requires no prior knowledge, and is capable of handling all cases.

4.2 Element Access and Assignment

JavaScript's element access and assignment operations are polymorphic and operate on various types of indexable data, such as arrays, strings and `TypedArray`s. This polymorphism restricts the applicability of optimizations; most of them can apply only when the type of the indexed data is known in advance.

Even when values are known to be arrays, JavaScript semantics invalidate common optimizations in the general case. For example, JavaScript does not require arrays in the C sense, that is, it does not require contiguous chunks of memory addressable by offset. Semantically, JavaScript arrays are plain objects that map indices – *string* representations of unsigned integers – to values. Element accesses into such arrays, then, are semantically (and perhaps surprisingly) equivalent to property lookups and are subject to the same set of rules, such as prototype lookups.

As with inferring object layout, SpiderMonkey attempts to infer when JavaScript arrays are used as if they were dense, C-like arrays, and optimize accordingly. Despite new APIs such as `TypedArray`s offering C-like arrays directly, SpiderMonkey's dense array optimizations remain crucial to the performance of the web.

To manage all possible modes of use of element accesses and the optimizations that apply in each of them, Ion relies on optimization tactics. We describe the most important optimization strategy – dense array access – below. The PIC and VM call cases are similar to the corresponding cases for property access. Other, specialized strategies heavily depend on

[13] The restriction on plain JavaScript objects is necessary because properties may be accessed from a variety of exotic object-like values, such as DOM nodes and proxies. Those objects encapsulate their own logic for accessing properties that is free to deviate from the logic perscribed for plain objects by the ECMAScript standard.

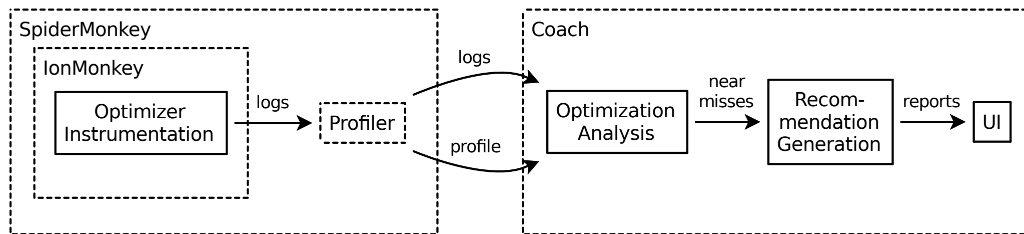


Figure 4 Our prototype's architecture.

SpiderMonkey's data representation and are beyond the scope of this paper, but are handled by the prototype.

Dense array access. Consider an element access `o[i]`. In the best case, if `o` is determined to be used as a dense array and `i` an integer, Ion can emit a memory load or a store for offset `i` plus bounds checking. For this choice to be valid, all types that flow into `o` must be plain JavaScript objects that have dense indexed properties. An object with few indexed properties spread far apart would be considered sparse, e.g., if only `o[0]` and `o[2048]` were set, `o` would not be considered dense. Note that an object may be missing indexed properties and still be considered dense. SpiderMonkey further distinguishes dense arrays – those with allocated dense storage – from packed arrays – dense arrays with no holes between indexed properties. Ion is able to elide checking whether an element is a hole, or a missing property, for packed arrays. Furthermore, the object `o` must not have been observed to have prototypes with indexed properties, as otherwise accessing a missing indexed property `j` on `o` would, per specification, trigger a full prototype walk to search for `j` when accessing `o[j]`.

5 Architecture

As Section 2.3 explains, our optimization coach operates in four phases. Figure 4 illustrates how these phases interact. In the first phase, instrumentation inside IonMonkey's optimizer logs optimization successes and failures and sends that information to the SpiderMonkey profiler (Section 7.1.1). Next, the optimization analysis phase applies pruning heuristics (sections 7.2.1 and 7.2.2), determines solution sites (Section 6.1), computes badness scores (Section 7.1.2), and finally merges reports (sections 6.3, 6.4, and 7.1.3). Its end result is a list of ranked near misses.

The third phase, recommendation generation, fills out textual recommendation templates – selected based on failure causes – with inferred solution sites, failure causes, type information, and source information. Finally, the tool's user interface presents the five highest-ranked recommendations to programmers.

6 Coaching for Object-Oriented Languages

Dispatch optimizations for property operations fundamentally depend on non-local information. For example, the optimizer must know the layout of objects that flow to a property access site to determine whether it can be optimized to a direct dereference. That information is encoded in the *constructor* of these objects, which can be arbitrarily far away in source text from the property access considered for optimization.

In turn, this gap causes optimization failures to be non-local; a failure at one program location – the property access site – can be resolved by program changes at a different

location – the constructor. To provide actionable feedback to programmers, a coach must connect the two sites and link its reports to the solution site.

Not all failures, however, are non-local in this manner. For example, failing to specialize a property access that receives multiple different types of objects is a purely local failure; it fails because the operation itself is polymorphic, which can only be solved by changing the operation itself. An optimization coach must therefore distinguish between local and non-local failures and target its reports accordingly. Our prototype accomplishes this using *solution site inference.*

In addition, a coach should also merge near misses that have the same, or similar, solutions and report them together. Our prototype uses *by-solution merging* and *by-constructor merging* for this purpose. It also reuses the notion of *locality merging* (see [22]).

6.1 Solution Site Inference

The goal of solution site inference is to determine, for a given near miss, whether it could be resolved by changes at the site of the failing optimization or whether changes to the receivers' constructors may be required. We refer to the former as *operation near misses* and to the latter as *constructor near misses.* To reach a decision, the coach follows heuristics based on the cause of the failure, as well as on the types that flow to the affected operation. We briefly describe two of these heuristics.

Monomorphic operations. If an optimization fails for an operation to which a single receiver type flows, then that failure must be due to a property of that type, not of the operation's context. The coach infers these cases to be constructor near misses.

Property addition. When a property assignment operation for property `p` receives an object that lacks a property `p`, the operation instead adds the property to the object. If the same operation receives both objects with a property `p` and objects without, that operation cannot be specialized for either mode of use. This failure depends on the operation's context, and the coach infers it to be an operation near miss.

6.2 Same-Property Analysis

The merging techniques we describe below both depend on grouping near misses that affect the same property. The obvious definitions of "same property," however, do not lead to satisfactory groupings. If we consider two properties with the same name to be the same, the coach would produce spurious groupings of unrelated properties from different parts of the program, e.g., grouping `canvas.draw` with `gun.draw`. Using these spurious groupings for merging would lead to incoherent reports that conflate unrelated near misses.

In contrast, if we considered only properties with the same name and the same hidden class, the coach would discriminate too much and miss some useful groupings. For example, consider the `run` property of various kinds of tasks in the Richards benchmark from the Octane[14] benchmark suite, boxed in Figure 5. These properties are set independently for each kind of task and thus occur on different hidden classes, but they are often accessed from the same locations and thus should be grouped by the coach. This kind of pattern occurs frequently when using inheritance or when using structural typing for ad-hoc polymorphism.

[14] `https://developers.google.com/octane/`

```
Scheduler.prototype.schedule = function () {
  // this.currentTcb is only ever a TaskControlBlock
  ...
  this.currentTcb = this.currentTcb.run();
  ...
};

TaskControlBlock.prototype.run = function () {
  // this.task can be all four kinds of tasks
  ...
  return this.task.run(packet);
  ...
};

IdleTask.prototype.run = function (packet) { ... };
DeviceTask.prototype.run = function (packet) { ... };
WorkerTask.prototype.run = function (packet) { ... };
HandlerTask.prototype.run = function (packet) { ... };
```

Figure 5 Two different logical properties with name `run` in the Richards benchmark, one underlined and one boxed.

To avoid these problems, we introduce another notion of property equivalence, *logical properties*, which our prototype uses to guide its near-miss merging. We define two concrete properties `p1` and `p2`, which appear on hidden classes `t1` and `t2` respectively, to belong to the same logical property if they

- have the same name `p`, and
- co-occur in at least one operation, i.e., there exists an operation `o.p` or `o.p = v` that receives objects of both class `t1` and class `t2`

As Figure 5 shows, the four concrete `run` properties for tasks co-occur at an operation in the body of `TaskControlBlock.prototype.run`, and therefore belong to the same logical property. `TaskControlBlock.prototype.run`, on the other hand, never co-occurs with the other `run` properties, and the analysis considers it separate; near misses that are related to it are unrelated from those affecting tasks' `run` properties and should not be merged.

6.3 By-Solution Merging

In addition to linking near-miss reports with the likely location of their solution, an optimization coach should group near misses with related solutions. That is, it should merge near misses that can be addressed either by same program change or by performing analogous changes at multiple program locations.

Detecting whether multiple near misses call for the same kind of corrective action is a simple matter of comparing the causes of the respective failures and their context, as well as ensuring that the affected properties belong to the same logical property. This mirrors the work of the recommendation generation phase, as described in Section 2.3.

Once the coach identifies sets of near misses with related solutions, it merges each set into a single summary report. This new report includes the locations of individual failures, as well as the common cause of failure, the common solution and a badness score that is the sum of those of the merged reports.

6.4 By-Constructor Merging

Multiple near misses can often be solved at the same time by changing a single constructor. For example, inconsistent property layout for objects from one constructor can cause optimization failures for multiple properties, yet all of those can be resolved by editing the constructor.

Therefore, merging constructor near misses that share a constructor can result in improved coaching reports.

To perform this merging, the coach must identify which logical properties co-occur within at least one hidden class. To do this, it reuses knowledge about which logical properties occur within each hidden class from same-property analysis.

Because, in JavaScript, properties can be added to objects dynamically – i.e., not inside the object's constructor – a property occuring within a given hidden class does not necessarily mean that it was added by the constructor associated with that class. This may lead to merging reports affecting properties added in a constructor with others added elsewhere. At first glance, this may appear to cause spurious mergings, but it is in fact beneficial. For example, moving property initialization from the outside of a constructor to the inside often helps keeping object layout consistent. Reporting these near misses along with those from properties from the constructor helps reinforce this connection. We discuss instances of this problem in Section 9.3.

7 Coaching for an Advanced Compiler

Advanced compilers such as IonMonkey operate differently from simpler compilers, such as the ahead-of-time portion of the Racket compiler, which we studied previously. An optimization coach needs to adapt to these differences. This section presents the challenges posed by two specific features of Ion that are absent in a simple compiler – JIT compilation and optimization tactics – and describes our solutions.

7.1 JIT Compilation

From a coaching perspective, JIT compilation poses two main challenges absent in an ahead-of-time (AOT) setting. First, compilation and execution are interleaved in a JIT system; there is no clear separation between compile-time and run-time, as there is in an AOT system. The latter's separation makes it trivial for a coach's instrumentation to not affect the program's execution; instrumentation, being localized to the optimizer, does not cause any runtime overhead and emitting the optimization logs does not interfere with the program's I/O proper. In a JIT setting, however, instrumentation may affect program execution, and a coach must take care when emitting optimization information.

Second, whereas an AOT compiler compiles a given piece of code once, a JIT compiler may compile it multiple times as it gathers more information and possibly revises previous assumptions. In turn, a JIT compiler may apply – and fail to apply – different optimizations each time. Hence, the near misses that affect a given piece of code may evolve over time, as opposed to being fixed, as in the case of an AOT compiler. Near misses therefore need to be ranked and merged along this new, temporal axis.

Our prototype coach addresses both challenges via the use of a novel, profiler-driven instrumentation strategy and by applying *temporal merging*, an extension of the locality merging technique we presented in previous work.

7.1.1 Profiler-Driven Instrumentation

Our prototype coach uses SpiderMonkey's profiling subsystem as the basis for its intrumentation. The SpiderMonkey profiler, as many profilers, provides an "event" API in addition to its main sampling-based API. The former allows the engine to report various kinds of one-off

events that may be of interest to programmers: Ion compiling a specific method, garbage collection, the execution bailing out of optimized code, etc.

This event API provides a natural communication channel between the coach's instrumentation inside Ion's optimizer and the outside world. As with an AOT coach, our prototype records optimization decisions and context information as the optimizer processes code. Where an AOT coach would emit that information on the fly, our prototype instead gathers all the information pertaining to a given invocation of the compiler, encodes it as a profiler event and emits it all at once. Our prototype's instrumentation executes only when the profiler is active; its overhead is therefore almost entirely pay-as-you-go.

In addition to recording optimization information, the instrumentation code assigns a unique identifier to the compiled code resulting from each Ion invocation. This identifier is included alongside the optimization information in the profiling event. Object code that is instrumented for profiling carries meta-information (e.g. method name and source location) that allows the profiler to map the samples it gathers back to source code locations. We include the compilation identifier as part of this meta-information, which allows the coach to correlate profiler samples with optimization information, which in turn enables heuristics based on profiling information as discussed below. This additional piece of meta-information has negligible overhead and is present only when the profiler is active.

7.1.2 Profiling-Based Badness Metric

One of the key advantages of an optimization coach over raw optimization logs is the pruning and ranking of near misses that a coach provides based on expected performance impact. An AOT coach uses a number of static heuristics to estimate this impact.

Our prototype incorporates profiling-based heuristics, which has two main advantages. First, even in an AOT setting, actionable prioritization of near misses benefits from knowing where programs spend their time; near misses in hot methods are likely to have a larger impact on performance than those in cold code.

Second, state-of-the-art JIT compilers may compile the same code multiple times – producing different *compiled versions* of that code – potentially with different near misses each time. A coach needs to know which of these compiled versions execute for a long time and which are short-lived. Near misses from compiled versions that execute only for a short time cannot have a significant impact on performance across the whole execution, regardless of the number or severity of near misses, or how hot the affected method is overall. Because profiler samples include compilation identifiers, our prototype associates each sample not only with particular methods, but with particular compiled versions of methods. It then enables the required distinctions discussed above.

Concretely, our prototype uses the *profiling weight* of the compiled version of the function that surrounds a near miss as its badness score. We define the profiling weight of a compiled version to be the fraction of the total execution time that is spent executing it. Combined with temporal merging, this design ensures that near misses from hot compiled versions rise to the top of the rankings.

To avoid overwhelming programmers with large numbers of potentially low-impact recommendations, our prototype prunes reports based on badness and shows only the five reports with the highest badness scores. This threshold has been effective in practice but is subject to adjustment.

7.1.3 Temporal Merging

Even though a JIT compiler may optimize methods differently each time they get compiled, this is not always the case. It is entirely possible to have an operation be optimized identically across multiple versions or even all of them. It happens, for instance, when recompilation is due to the optimizer's assumptions not holding for a different part of the method or as a result of object code being garbage collected.[15]

Identical near misses that originate from different invocations of the compiler necessarily have the same solution; they are symptoms of the same underlying issue. To reduce redundancy in the coach's reports, we extend the notion of locality merging – which merges reports that affect the same operation – to operate across compiled version boundaries. The resulting technique, *temporal merging*, combines near misses that affect the same operation or constructor, originate from the same kind of failure and have the same causes across multiple compiled versions.

7.2 Optimization Tactics

When faced with an array of optimization options, Ion relies on optimization tactics to organize them. While we could consider each individual element of a tactic as a separate optimization and report near misses accordingly, all of a tactic's elements are linked. Because the entire tactic returns as soon as one element succeeds, its options are mutually exclusive; only the successful option applies. To avoid overwhelming programmers with multiple reports about the same operation and provide more actionable results, a coach should consider a tactic's options together.

7.2.1 Irrelevant Failure Pruning

Ion's tactics often include strategies that only apply in narrow cases – e.g. indexing into values that are known to be strings, property accesses on objects that are known to be constant, etc. Because of their limited applicability, failure to apply these optimizations is not usually symptomatic of performance issues; these optimizations are expected to fail most of the time.

In these cases, we reuse the Racket coach's *irrelevant failure pruning* technique. Failures to apply optimizations that are expected to fail do not provide any actionable information to programmers, and thus we consider them irrelevant. The coach prunes such failures from the logs and does not show them in its reports.

7.2.2 Partial Success Shortcircuiting

While some elements of a given tactic may be more efficient than others, it is not always reasonable to expect that all code be compiled with the best tactic elements. For example, polymorphic call sites cannot be optimized as well as monomorphic call sites; polymorphism notably prevents fixed-slot lookup. Polymorphism, however, is often desirable in a program. Recommending that programmers eliminate it altogether in their programs is preposterous

[15] In SpiderMonkey, object code is collected during major collections to avoid holding on to object code for methods that may not be executed anymore. While such collections may trigger more recompilation than strictly necessary, this tradeoff is reasonable in the context of a browser, where most scripts are short-lived.

and would lead to programmers ignoring the tool. Clearly, considering all polymorphic operations to suffer from near misses is not effective.

We partition a tactic's elements according to source-level concepts – e.g., elements for monomorphic operations vs polymorphic operations, elements that apply to array inputs vs string inputs vs typed array inputs, etc. – and consider picking the best element from a group to be an optimization success, so long as the operation's context matches that group.

For example, the coach considers picking the best possible element that is applicable to polymorphic operations to be a success, as long as we can infer from the context that the operation being compiled is actually used polymorphically. Any previous failures to apply monomorphic-only elements to this operation would be ignored.

With this approach, the coach reports polymorphic operations that do not use the best possible polymorphic element as near misses, while considering those that do to be successes. In addition, because the coach considers only uses of the best polymorphic elements to be successes if operations are actually polymorphic according to their context, monomorphic operations that end up triggering them are reported as near misses – as they should be.

In addition to polymorphic property operations, our prototype applies partial success shortcircuiting to array operations that operate on typed arrays and other indexable datatypes. For example, Ion cannot apply dense-array access for operations that receive strings, but multiple tactic elements can still apply in the presence of strings, some more performant than others.

8 Dead Ends

The previous sections describe successful coaching techniques, which result in actionable reports. Along the way, we also implemented other techniques that ultimately did not prove to be useful and which we removed from our prototype. These techniques either produced reports that did not lead programmers to solutions or pointed out optimization failures that did not actually impact performance.

In the interest of saving other researchers from traveling down the same dead ends, this section describes two kinds of optimization failures that we studied without success: *regressions* and *flip-flops*. Both are instances of *temporal patterns*, that is, attempts by the coach to find optimization patterns across time. None of our attempts at finding such patterns yielded actionable reports, but there may be other kinds of temporal patterns that we overlooked that would.

8.1 Regression Reports

The coach would report a regression when an operation that was optimized well during a compilation failed to be optimized as well during a subsequent one. This pattern occurred only rarely in the programs we studied, and when it did, it either was inevitable (e.g. a call site becoming polymorphic as a result of observing a sentinel value in addition to its usual receiver type) or did not point to potential improvements.

8.2 Flip-Flop Reports

As mentioned, SpiderMonkey discards object code and all type information during major collections. When this happens, the engine must start gathering type information and compiling methods from scratch. In some cases, this new set of type information may lead the engine to be optimistic in a way that was previously invalidated, then forgotten during

garbage collection, leading to excessive recompilation. Engine developers refer to this process of oscillating back and forth between optimistic and conservative versions as *flip-flopping*.

For example, consider a method that almost always receives integers as arguments, but sometimes receives strings as well. Ion may first optimize it under the first assumption, then have to back out of this decision after receiving strings. After garbage collection, type information is thrown away and this process starts anew. As a result, the method may end up being recompiled multiple times between each major collection.

Engine developers believe that this behavior can cause significant performance issues, mostly because of the excessive recompilation. While we observed instances of flip-flopping in practice, modifying the affected programs to eliminate these recompilations often required significant reengineering and did not yield observable speedups.

9 Evaluation

For an optimization coach to be useful, it must provide actionable recommendations that improve the performance of a spectrum of programs. This section shows the results of evaluating our prototype along two axes: performance improvements and programmer effort.

9.1 Experimental Protocol

For our evaluation, we chose a subset of the widely-used Octane benchmark suite. We ran these programs using our prototype and modified them by following the coach's recommendations. For each program, we applied all of the five top-rated recommendations, so long as the advice was directly actionable. That is, we rejected reports that did not suggest a clear course of action, as a programmer using the tool would do.

To simulate a programmer looking for "low-hanging fruit," we ran the coach only once on each program. Re-running the coach on a modified program may cause the coach to provide different recommendations. Therefore, it would in principle be possible to apply recommendations up to some fixpoint.

For each program and recommendation, we measured a number of attributes to assess three dimensions of optimization coaching:

Performance Impact. Our primary goal is to assess the effect of recommended changes on program performance. Because a web page's JavaScript code is likely to be executed by multiple engines, we used three of the major JavaScript engines: SpiderMonkey, Chrome's V8 and Webkit's JavaScriptCore.

The Octane suite measures performance in terms of an *Octane Score* which, for the benchmarks we discuss here, is inversely proportional to execution time.[16] Our plots show scores normalized to the pre-coaching version of each program with error bars marking 95% confidence intervals. All our results represent the mean score of 30 executions on a 6-core 64-bit x86 Debian GNU/Linux system with 12GB of RAM. To eliminate confounding factors due to interference from other browser components, we ran our experiments in standalone JavaScript shells.

[16] The Octane suite also includes benchmarks whose scores are related to latency instead of execution time, but we did not use those for our experiments.

Programmer Effort. As a proxy for programmer effort, we measured the number of lines changed in each program while following recommendations. We also recorded qualitative information about the nature of these changes.

Recommendation Usefulness. To evaluate the usefulness of individual recommendations, we classified them into four categories:

- *positive* recommendations led to an increase in performance,
- *negative* recommendations led to a decrease in performance,
- *neutral* recommendations did not lead to an observable change in performance, and
- *non-actionable* reports did not suggest a clear course of action.

For this aspect of the evaluation, we measured the impact of individual recommendations under SpiderMonkey alone.

Ideally, a coach should give only positive recommendations. Negative recommendations require additional work on the part of the programmer to identify and reject. Reacting to neutral recommendations is also a waste of programmer time, and thus their number should be low, but because they do not harm performance, they need not be explicitly rejected by programmers. Non-actionable recommendations decrease the signal-to-noise ratio of the tool, but they can individually be dismissed pretty quickly by programmers. A small number of non-actionable recommendations therefore does not contribute significantly to the programmer's workload. Large numbers of non-actionable recommendations, however, would be cause for concern.

9.2 Program Selection

Our subset of the Octane suite focuses on benchmarks that use property and array operations in a significant manner. It excludes, for example, the Regexp benchmark because it exercises nothing but an engine's regular expression subsystems. Coaching these programs would not yield any recommendations with our current prototype. It also excludes machine-generated programs from consideration. The output of, say, the Emscripten C/C++ to JavaScript compiler[17] is not intended to be read or edited by humans; it is therefore not suitable for coaching.[18] In total, the set consists of eight programs: Richards, DeltaBlue, RayTrace, Splay, NavierStokes, PdfJS, Crypto and Box2D.

9.3 Results and Discussion

As Figure 6 shows, following the coach's recommendations leads to significant[19] speedups on six of our eight benchmarks when run on SpiderMonkey. These speedups range from 1.02× to 1.17×. For the other two benchmarks, we observe no significant change; in no case do we observe a slowdown.

The results are similar for the other engines, see Figure 7. On both V8 and JavaScriptCore, we observe significant speedups on two and three benchmarks, respectively, ranging from 1.02× to 1.20×. These speedups differ from those observed using SpiderMonkey, but are of similar magnitude. Only in the case of the DeltaBlue benchmark on JavaScriptCore is

[17] `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Emscripten`

[18] It would, however, be possible to use coaching to improve the code generation of Emscripten or other compilers that target JavaScript, such as Shumway. This is a direction for future work.

[19] We consider speedups to be significant when the confidence intervals of the baseline and coached versions do not overlap.

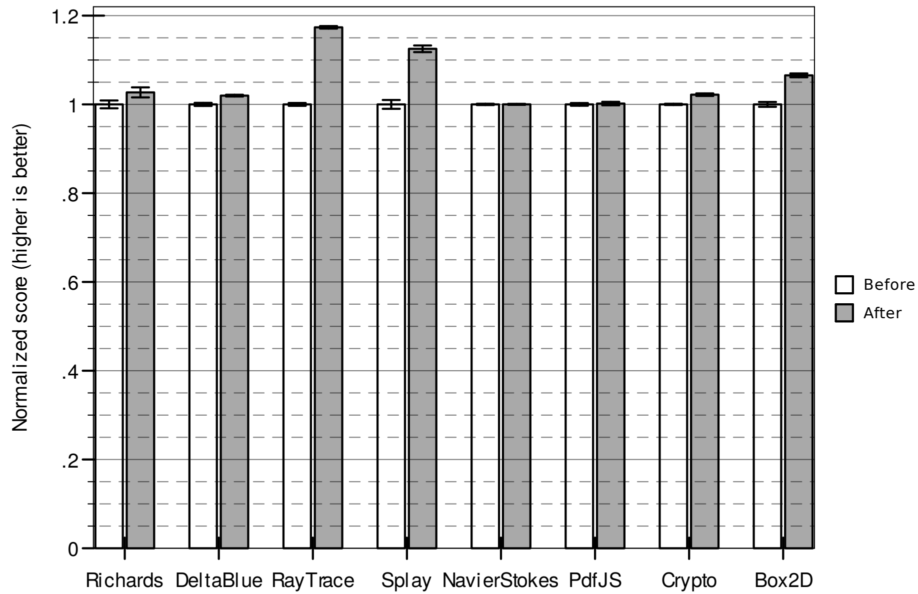


Figure 6 Benchmarking results on SpiderMonkey.

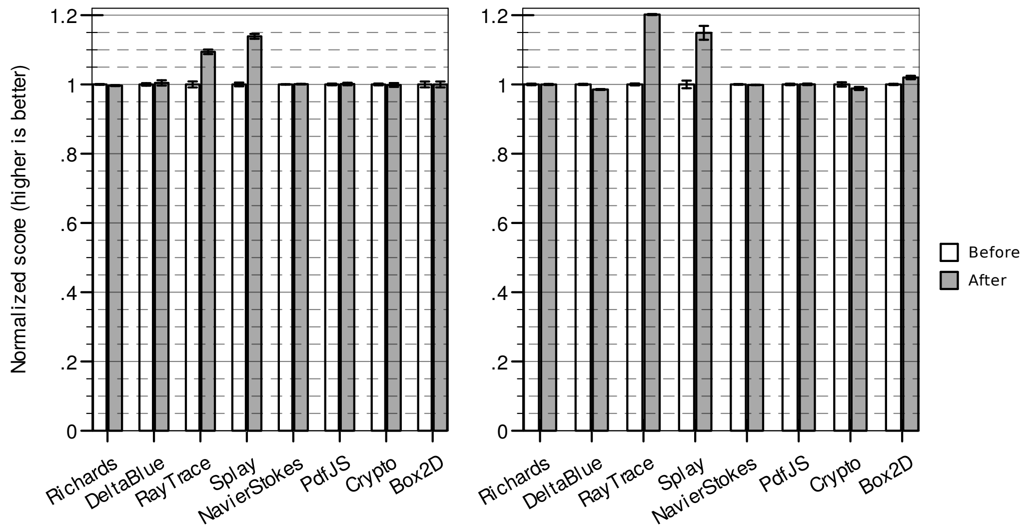


Figure 7 Benchmarking results on V8 and JavaScriptCore.

Benchmark	Size (SLOC)	Lines changed (SLOC)			Recommendation impact (# recommendations)			
		Added	Deleted	Edited	Positive	Negative	Neutral	Non-act.
Richards	538	1	5	0	2	0	0	1
DeltaBlue	881	12	6	24	2	1	1	1
RayTrace	903	10	11	0	5	0	0	0
Splay	422	3	3	0	2	0	1	2
NavierStokes	415	0	0	4	0	0	1	0
PdfJS	33,053	2	1	0	0	0	1	4
Crypto	1,698	2	0	1	4	0	0	1
Box2D	10,970	8	0	0	2	0	0	3

Figure 8 Summary of changes following recommendations.

there a significant slowdown. These results provide evidence that, even though coaching recommendations are derived from the optimization process of a *single engine*, they can lead to *cross-engine* speedups.

Keeping in mind that JavaScript engines are tuned to perform well on those benchmark programs,[20] we consider these results quite promising. We conjecture that our prototype (or an extension of it) could yield even larger speedups on other, regular programs for which the engine is not specifically tuned.

Figure 8 presents our results for the effort and usefulness dimensions. For all programs, the total number of lines changed is at most 42. Most of these changes are also fairly mechanical in nature – moving code, search and replace, local restructuring. Together, these amount to modest efforts on the programmer's part.

We classified 17 out of 35 reports as positive, and only one as negative. We classified 12 reports as non-actionable, which we consider acceptably low. As discussed above, those reports can be dismissed quickly and do not impose a burden. The remainder of the section presents the coach's recommendations for individual benchmarks.

Richards. The coach provides three reports. Two of those point out an inconsistency in the layout of `TaskControlBlock` objects. Figure 9 shows one of them. The `state` property is initialized in two different locations, which causes layout inference to fail and prevents optimizations when retrieving the property. Combining these two assignments into one, as Figure 10 shows, solves the issue and leads to a speedup of 1.03× on SpiderMonkey. The third report points to an operation that is polymorphic by design; it is not actionable.

DeltaBlue. Two of the five reports have a modest positive impact. The first involves replacing a singleton object's properties with global variables to avoid dispatch; it is shown in Figure 11. The second recommends duplicating a superclass's method in its subclasses, making them monomorphic in the process.

These changes may hinder modularity and maintainability in some cases. They clearly illustrate the tradeoffs between performance and software engineering concerns, which coaching tends to bring up. Which of those is more important depends on context, and the decision of whether to follow a recommendation must remain in the programmer's hands.

[20] `http://arewefastyet.com`

```
badness: 24067
for object type: TaskControlBlock:richards.js:255

affected properties:
    state (badness: 24067)

This property is not guaranteed to always be in the same location.

Are properties initialized in different orders in different places?
  If so, try to stick to the same order.
Is this property initialized in multiple places?
  If so, try initializing it always in the same place.
Is it sometimes on instances and sometimes on the prototype?
  If so, try using it consistently.
```

Figure 9 Report of inconsistent property order in the Richards benchmark.

```
// before coaching
if (queue == null) {
  this.state = STATE_SUSPENDED;
} else {
  this.state = STATE_SUSPENDED_RUNNABLE;
}

// after coaching
this.state = queue == null ? STATE_SUSPENDED : STATE_SUSPENDED_RUNNABLE;
```

Figure 10 Making object layout consistent in the Richards benchmark.

```
badness: 5422
for object type: singleton
affected properties:
    WEAKEST (badness: 2148)
    REQUIRED (badness: 1640)
    STRONG_DEFAULT (badness: 743)
    PREFERRED (badness: 743)
    NORMAL (badness: 147)

This object is a singleton.
Singletons are not guaranteed to have properties in a fixed slot.

Try making the object's properties globals.
```

Figure 11 Recommendation to eliminate a singleton object in the DeltaBlue benchmark.

With a coach, programmers at least know *where* these tradeoffs may pay off by enabling additional optimization.

One of the recommendations (avoiding a prototype chain walk) yields a modest slowdown of about 1%. This report has the lowest badness score of the five. We expect programmers tuning their programs to try out these kinds of negative recommendations and revert them after observing slowdowns.

RayTrace. All five of the coach's reports yield performance improvements, for a total of 1.17× on SpiderMonkey, 1.09× on V8 and 1.20× on JavaScriptCore. The proposed changes include reordering property assignments to avoid inconsistent layouts, as well as replacing a use of prototype.js's class system with built-in JavaScript objects for a key data structure. All these changes are mechanical in nature because they mostly involve moving code around.

Splay. This program is the same as the example in Section 2.2 Of the five reports, three recommend moving properties from a prototype to its instances. These properties are using a default value on the prototype and are sometimes left unset on instances, occasionally triggering prototype chain walks. The fix is to change the constructor to assign the default value to instances explicitly. While this may cause additional space usage by making instances larger, the time/space tradeoff is worthwhile and leads to speedups on all three engines. Two of the three changes yield speedups, with the third one not having a noticeable effect.

NavierStokes. The coach provides a single recommendation for this program. It points out that some array accesses are not guaranteed to receive integers as keys. Enforcing this guarantee by bitwise or'ing the index with `0`, as is often done in asm.js codebases, solves this issue but does not yield noticeable performance improvements. It turns out that the code involved only accounts for only a small portion of total execution time.

PdfJS. One of the coach's reports recommends initializing two properties in the constructor, instead of waiting for a subsequent method call to assign them, because the latter arrangement results in inconsistent object layouts. As with the recommendation for the NavierStokes benchmark, this one concerns cold code[21] and does not lead to noticeable speedups.

We were not able to make changes based on the other four recommendations, which may have been due to our lack of familiarity with this large codebase. Programmers more familiar with PdfJS's internals may find these reports more actionable.

Crypto. Four of the five reports are actionable and lead to speedups. Three of the four concern operations that sometimes add a property to an object and sometimes assign an existing one, meaning that they therefore cannot be specialized for either use. Initializing those properties in the constructor makes the above operations operate as assignments consistently, which solves the problem. The last positive recommendation concerns array accesses; it is similar to the one discussed in conjunction with the NavierStokes benchmark, with the exception that this one yields speedups.

[21] PdfJS's profile is quite flat in general, suggesting that most low-hanging fruit has already been picked, which is to be expected from such a high-profile production application.

Box2D. Two of the reports recommend consistently initializing properties, as with the PdfJS benchmark. Applying those changes yields a speedup of 1.07× on SpiderMonkey. The other three recommendations are not actionable due to our cursory knowledge of this codebase. As with PdfJS, programmers knowledgeable about Box2D's architecture may fare better.

For reference, the Octane benchmark suite uses a minified version of this program. As discussed above, minified programs are not suitable for coaching so we used a non-minified, but otherwise identical, version of the program.

10 Related Work

This work is not the only attempt at helping programmers take advantage of their compilers' optimizers. This section discusses tools with similar goals and compares them with our work.

10.1 Optimization Logging

From an implementation perspective, the simplest way to inform programmers about the optimizer's behavior on their programs is to provide them with logs recording its optimization decisions. This is the approach taken by tools such as JIT inspector [12] and IRHydra [8], both of which report optimization successes and failures, as well as other optimization-related events such as dynamic deoptimizations. JIT inspector reports optimizations performed by IonMonkey, while IRHydra operates with the V8 and Dart compilers.

Similar facilities also exist outside of the JavaScript world. For instance, Common Lisp compilers such as SBCL [23] and LispWorks [18] report both optimization successes and optimization failures, such as failures to specialize generic operations or to allocate objects on the stack. The Cray XMT C and C++ compilers [4] report both successful optimimizations and parallelization failures. The Open Dylan IDE [6, chapter 10] reports optimizations such as inlining and dispatch optimizations using highlights in the IDE's workspace.

These tools provide reports equivalent to the raw output of our prototype's instrumentation without any subsequent analysis, interpretation or recommendations. Expert programmers knowledgeable about compiler internals may find this information actionable and use it as a starting point for their tuning efforts. In constrast, our prototype coach targets programmers who may not have the necessary knowledge and expertise to digest such raw information, and it does so by providing recommendations that only require source-level knowledge.

10.2 Rule-Based Performance Bug Detection

Some performance tools use rule-based approaches to detect code patterns that may be symptomatic of performance bugs.

JITProf [10] is a dynamic analysis tool for JavaScript that detects code patterns that JavaScript JIT compilers usually do not optimize well. The tool looks for six dynamic patterns during program execution, such as inconsistent object layouts and arithmetic operations on the `undefined` value, and reports instances of these patterns to programmers.

The JITProf analysis operates independently from the host engine's optimizer; its patterns essentially constitute a model of a typical JavaScript JIT compiler. As a result, JITProf does not impose any maintenance burden on engine developers, unlike a coach whose instrumentation must live within the engine itself. Then again, this separation may cause the tool's model to be inconsistent with the actual behavior of engines, either because the model does not perfectly match an engine's heuristics, or because engines may change their

optimization strategies as their development continues. In contrast, an optimization coach reports ground truth by virtue of getting its optimization information from the engine itself.

By not being tied to a specific engine, JITProf's reports are not biased by the implementation details of that particular engine. Section 9 shows, however, that engines behave similarly enough in practice that a coach's recommendations, despite originating from a specific engine, usually lead to cross-engine performance improvements.

Jin et al. [16] distill performance bugs found in existing applications to source-level patterns which can then be used to detect similar latent bugs in other applications. Their tool suggests fixes for these new bugs based on those used to resolve the original bugs. Their work focuses on API usage and algorithms, and is complementary to optimization coaching.

Chen et al. [3] present a tool that uses static analysis to detect performance anti-patterns that result from the use of object-relational mapping in database-backed applications. The tool detects these anti-patterns using rules that the authors synthesized from observing existing database-related performance bugs. To cope with the large number of reports, the tool estimates the performance impact of each anti-pattern, and uses that information to prioritize reports. This is similar to the use of ranking by optimization coaches.

10.3 Profilers

When they encounter performance issues, programmers often reach for a profiler [11, 19, 21, 24]. Unlike an optimization coach, a profiler does not point out optimization failures directly. Instead, it identifies portions of the program where most of its execution time is spent, some of which may be symptomatic of optimization failures. That inference, however, is left to programmers.

Profilers also cannot distinguish between code that naturally runs for a long time from code that runs for an abnormally long time. Again, the programmer is called upon to make this distinction. In contrast, coaches distinguish between optimization failures that are expected from those that are not. In addition, coaches aim to provide actionable recommendations to programmers, whereas profilers report data without pointing towards potential solutions.

Note, though, that profilers can point to a broader range of performance issues than optimization coaches. For example, a profiler would report code that runs for a long time due to an inefficient algorithm, which an optimization coach could not detect. To summarize, the two classes of tools cover different use cases and are complementary.

10.4 Assisted Optimization

A number of performance tools are aimed at helping programmers optimize specific aspects of program performance. This section discusses the ones most closely related to this work.

Larsen et al. [17] present an interactive tool that helps programmers parallelize their programs. Like an optimization coach, their tool relies on compiler instrumentation to reconstruct the optimization process – specifically automatic parallelization – and discover the causes of parallelization failures. Larsen et al.'s tool is specifically designed for parallelization and is thus complementary to optimization coaching.

Precimonious [20] is a tool that helps programmers balance precision and performance in floating-point computations. It uses dynamic program analysis to discover floating-point variables that can be converted to use lower-precision representations without affecting the overall precision of the program's results. The tool then recommends assignments of precisions to variables that programmers can apply. This workflow is similar to that of an optimization coach, but applied to a different domain.

Xu et al. [25] present a tool that detects data structures that are expensive to compute, but that the program either does not use, or only uses a small portion of. Based on the tool's reports, programmers can replace the problematic structures with more lightweight equivalents that only store the necessary data. The tool relies on a novel program slicing technique to detect those low-utility data structures. This tool is also complementary to optimization coaches.

11 Conclusion

In this paper, we present an adaptation of optimization coaching to the world of dynamic object-oriented languages with advanced JIT compilers. The additional constraints imposed by these languages and their compilers require novel coaching techniques such as profiler-based instrumentation and solution-site inference.

We additionally provide evidence, in the form of case studies using well-known benchmark programs, that optimization coaching is an effective means of improving the performance of JavaScript programs. The evaluation also shows that its usage is well within the reach of JavaScript programmers.

Acknowledgment. We would like to thank Niko Matsakis, Dave Herman, and Michael Bebenita for discussions and suggestions about the tool's design and development. Kannan Vijayan, Luke Wagner, and Nicolas Pierron helped with the design of the profiler-driven instrumentation. Finally, we thank Matthias Felleisen, Sam Tobin-Hochstadt and Jan Vitek for their comments on previous drafts.

This work was partially supported by Darpa, NSF SHF grants 1421412, 1421652, and Mozilla.

References

1. Craig Chambers and David Ungar. Iterative type analysis and extended message splitting. *Lisp and Symbolic Computation* 4(3), pp. 283–310, 1990.
2. Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF. In *Proc. OOPSLA*, pp. 49–70, 1989.
3. Tse-Hun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohammed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proc. ICSE*, pp. 1001–1012, 2014.
4. Cray inc. *Cray XMT™ Performance Tools User's Guide*. 2011.
5. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS* 13(4), pp. 451–490, 1991.
6. Dylan Hackers. *Getting Started with the Open Dylan IDE*. 2015. `http://opendylan.org/documentation/getting-started-ide/GettingStartedWithTheOpenDylanIDE.pdf`
7. ECMA International. *ECMAScript® Language Specification*. Standard ECMA-262, 2011.
8. Vyacheslav Egorov. *IRHydra Documentation*. 2014. `http://mrale.ph/irhydra/`
9. Matthew Flatt and PLT. Reference: Racket. PLT Inc., PLT-TR-2010-1, 2010. `http://racket-lang.org/tr1/`
10. Liang Gong, Michael Pradel, and Koushik Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. University of California at Berkeley, UCB/EECS-2014-144, 2014.
11. Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: a call graph execution profiler. In *Proc. Symp. on Compiler Construction*, pp. 120–126, 1982.

12 Brian Hackett. *JIT Inspector Add-on for Firefox*. 2013. `https://addons.mozilla.org/en-US/firefox/addon/jit-inspector/`
13 Brian Hackett and Shu-yu Guo. Fast and precise type inference for JavaScript. In *Proc. PLDI*, pp. 239–250, 2012.
14 Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. ECOOP*, pp. 21–38, 1991.
15 Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proc. PLDI*, pp. 32–43, 1992.
16 Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proc. PLDI*, pp. 77–88, 2012.
17 Per Larsen, Razya Ladelsky, Jacob Lidman, Sally A. McKee, Sven Karlsson, and Ayal Zaks. Parallelizing more loops with compiler guided refactoring. In *Proc. International Conf. on Parallel Processing*, pp. 410–419, 2012.
18 LispWorks Ltd. *LispWorks© 6.1 Documentation*. 2013.
19 Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the accuracy of Java profilers. In *Proc. PLDI*, pp. 187–197, 2010.
20 Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: Tuning assistant for floating-point precision. In *Proc. Conf. for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2013.
21 Aibek Sarimbekov, Andreas Sewe, Walter Binder, Philippe Moret, and Mira Mezini. JP2: Call-site aware calling context profiling for the Java virtual machine. *SCP* 79(EST 4), pp. 146–157, 2014.
22 Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching: optimizers learn to communicate with programmers. In *Proc. OOPSLA*, pp. 163–178, 2012.
23 The SBCL Team. *SBCL 1.0.55 User Manual*. 2012.
24 Guoqing Xu. Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs. In *Proc. OOPSLA*, pp. 111–130, 2013.
25 Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *Proc. PLDI*, pp. 174–186, 2010.

PerfBlower: Quickly Detecting Memory-Related Performance Problems via Amplification

Lu Fang[1], Liang Dou*[2], and Guoqing Xu[1]

1 **University of California, Irvine, USA**
{lfang3, guoqingx}@ics.uci.edu

2 **East China Normal University, China**
ldou@cs.ecnu.edu.cn

Abstract

Performance problems in managed languages are extremely difficult to find. Despite many efforts to find those problems, most existing work focuses on how to debug a user-provided test execution in which performance problems already manifest. It remains largely unknown how to effectively find performance bugs before software release. As a result, performance bugs often escape to production runs, hurting software reliability and user experience. This paper describes PerfBlower, a general performance testing framework that allows developers to quickly test Java programs to find *memory-related* performance problems. PerfBlower provides (1) a novel specification language ISL to describe a general class of performance problems that have observable symptoms; (2) an automated test oracle via *virtual amplification*; and (3) precise reference-path-based diagnostic information via *object mirroring*. Using this framework, we have amplified three different types of problems. Our experimental results demonstrate that (1) ISL is expressive enough to describe various memory-related performance problems; (2) PerfBlower successfully distinguishes executions with and without problems; *8 unknown problems* are quickly discovered under small workloads; and (3) PerfBlower outperforms existing detectors and does not miss any bugs studied before in the literature.

1998 ACM Subject Classification D.3.4 [Programming Languages] Processors – Memory management, optimization, run-time environments, F.3.2 [Logics and Meaning of Programs] Semantics of Programming Languages – Program analysis, D.2.5 [Software Engineering] Testing and Debugging – Debugging aids

Keywords and phrases Performance bugs, memory problems, managed languages, garbage collection

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.296

1 Introduction

Performance problems commonly exist in real-world applications. Much evidence [13, 35, 19] shows that seemingly insignificant performance problems can lead to severe scalability reductions and even financial losses. Performance problems are notoriously difficult to find and fix because visible performance degradation is often an accumulation of the effects of a great number of invisible problems that scatter all over the program [35]. These problems all seem harmless unless they are triggered together under a large workload, wherein their effects multiply and manifest, preventing the program from reaching its performance/scalability goals.

* Work was done while the author visited UC Irvine during 2013–2014.

© Lu Fang, Liang Dou, and Guoqing Xu;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 296–320

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Most existing research efforts that attempt to find performance problems are postmortem debugging techniques [13, 22, 29, 12, 34, 42], focused on detecting the root cause of a performance problem that already manifests in a user-provided test execution or actual production run. However, in performance testing, developers often execute a large number of tests to achieve coverage, and, thus, it is unrealistic to ask developers to run a detector on each test and check if each generated report (often with hundreds of warnings) contains true performance bugs. This is a task even harder than finding a needle in a haystack because whether there exists a needle is unknown in the first place, let alone how to search the haystack to find it. Symptom-based detectors are more suitable for *postmortem debugging* when a performance bug already manifests in a user-provided test – developers are much more willing to spend their time digging into reported warnings if they already have some initial clue on the bug.

In addition to these research efforts, many open-source frameworks (e.g., [15, 18, 30, 31, 17]) have been designed to support performance testing, e.g., by generation of large tests/loads for triggering performance bugs[1]. However, these existing performance testing frameworks suffer from the following three major drawbacks: (1) the lack of a general specification to describe performance problems; (2) the lack of effective test oracles that can capture invisible performance problems under small (testing) workloads; checks on the absolute time and/or memory are often very subjective and cannot reveal small performance bugs before they accumulate; and (3) the lack of effective debugging support that can help developers find the root cause when a bug manifests. As a result, performance bugs still escape to production runs, hurt user experience, degrade system throughput, and waste computational resources [6]. They affect even well tested software such as Windows 7's Windows Explorer, which had several high-impact performance bugs that escaped detection for long periods of time [12].

Our target. In this paper, we propose a general performance testing framework, called PerfBlower, that can "blow up" the effect of small performance problems so that they can be easily captured during testing. Finding general performance problems is infeasible, as it requires finding an alternative, more efficient computation in a possibly infinite solution space. We first narrow our focus onto a class of performance problems that have *observable symptoms*; their symptoms can be expressed by logical statements over a history of heap updates. These problems include, to name a few, memory leaks, inefficiently-used collections, unused return values, or loop-invariant data structures. One common axis that centers around these problems is that they manifest in the form of inefficient data usage, and their symptoms can be identified by capturing and comparing heap snapshots. Many performance problems are caused by redundant computations; although they are not directly data-related, data inefficiencies can still be seen as a result of redundant computations. For example, one problem in Mozilla studied in [13] is due to the over-general implementation of an API – the `draw` method performs heavy-weight computations to draw high-quality images while the client only needs transparent ones. Even if the proposed technique cannot directly amplify redundant computations, most fields of the resulting image object are never used; these redundant data can be captured and penalized.

Contribution 1: A performance specification language. To provide developers a general way to define symptoms, we propose a simple, event-based language, referred to as ISL

[1] We use "performance bug" and "performance problem" interchangeably.

(i.e., acronym for *instrumentation specification language*). ISL is a domain-specific language tailored for describing symptoms of performance problems on a JVM. Since performance problems cannot be expressed using logical assertions, an important challenge in the design of ISL is what to do when a symptom is seen. Unlike a regular detector that immediately reports a problem, ISL provides a pair of commands *amplify* and *deamplify*, which allow developers to add *memory penalties* to an object upon observing a symptom and remove penalties when a *counter-evidence* is seen. Symptoms are only *indicators* of performance problems, not *definitive evidence*. If the specified symptom does not point to a real problem (e.g., the symptom stays for a while and then disappears), the developer calls *deamplify* to remove penalties. The symptom specified in ISL is periodically checked (often during garbage collection): objects that keep satisfying the specification will keep getting penalized; all existing penalties for an object are removed at the moment we observe that the object does not satisfy the specification.

Contribution 2: Providing test oracle via virtual amplification. Our amplification is on a *per-object* basis: a memory penalty is created and associated with each object that satisfies the symptom specified in ISL (i.e., at the moment the object becomes "suspicious").

Instead of requesting actual memory as penalties, which would incur significant space overhead, we create *virtual penalties* by maintaining a *penalty counter* (PC) inside each object to track the size of the penalty created for the object. During each garbage collection (GC), we identify the real heap usage of the program and then compute a *virtual heap consumption* by adding up the PC for each object and the real heap consumption. The virtual heap consumption is then compared to the real heap consumption to compute a *virtual space overhead* (VSO). VSO provides an automated test oracle: our experimental results show that the overall VSOs for benchmarks with and without real performance problems are, respectively, 20+ times and 1.5 times. The gap is sufficiently large so that test runs triggering bugs can be easily identified.

Contribution 3: Providing precise diagnostic information via object mirroring. PERFBLOWER is not only able to amplify the effects of performance problems; it can also provide diagnostic information to help developers find root causes of these problems. Much evidence [36, 41] shows object reference path is an important piece of information that reveals both the calling context and the data structure in which a "suspicious" object is created. We propose a novel algorithm (Section 4) that *incrementally* builds a *mirror object chain* that reflects the major reference path in the object graph leading to *o*. This chain records the source code information of each object on the reference path; identifying and reporting the reference path for *o* gets reduced to traversing *forward o*'s *mirror chain*, a task significantly easier than performing a *backward* traversal on the object graph which has only unidirectional edges.

Our incremental algorithm enables an important feature of the framework: *the completeness of diagnostic information provided for a "suspicious" object is proportional to the degree of its suspiciousness* (i.e., how long it gets penalized). Every time the object is penalized, the algorithm presented in Section 4 incrementally records one additional level of the reference path for the object. While a regular detector can also provide diagnostic information, it maintains a tracking space of the same size and records the same amount of information for all objects. Developers are very often interested in only a few (top) warnings in a diagnostic report; hence, it is much better to record more diagnostic information for top warnings than those that are low down on the list. Our algorithm dynamically determines the metadata

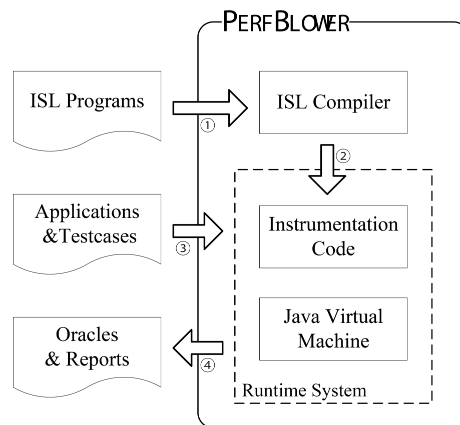


Figure 1 The architecture of PERFBLOWER.

space size based on how "interesting" an object is, making it possible to *prioritize* object tracking. For example, the reference path associated with the top object in `xalan` (shown in Section 5.2) reported by our leak amplifier has 10 nodes; all of the nodes on this long path are necessary for us to understand the cause of the leak. A regular detector can never afford to store a 10-node path for all objects in the heap.

We have implemented PERFBLOWER based on the 3.1.3 version of the Jikes Research Virtual Machine (RVM). Using the framework, we have amplified three different types of performance problems (i.e., memory leaks, under-utilized containers, and over-populated containers) on a set of real-world applications; our experimental results show that even under small workloads, there is a very clear distinction between the VSOs for executions with and without problems, making it particularly easy for developers to write simple space assertions and only inspect programs that have assertion failures. In fact, we have manually inspected each of our benchmarks for which PERFBLOWER reports a high VSO, and found a total of *8 unknown problems*.

We have compared the quality of leak reports between PERFBLOWER and an existing leak detector SLEIGH [3] executed under the same (small) workloads: PERFBLOWER reported 7 leaks with no false positive and very detailed diagnostic information, while SLEIGH reported 5 leaks with 1 false positive, and generated zero diagnostic information for 3 programs and very high-level information for the other 2 programs. We additionally performed an exhaustive study of the 14 performance bugs reported in the literature: PERFBLOWER found all but 4 bugs; for these bugs, we either could not run the program or did not have a triggering test case. These promising results clearly show that PERFBLOWER is useful in quickly building a large number of checkers that can effectively find performance bugs before they manifest.

2 PERFBLOWER Overview

The PERFBLOWER architecture. Figure 1 depicts the architecture of PERFBLOWER. It has two major components: an ISL compiler and a runtime system. The compiler parses an ISL program provided by the developer and automatically generates instrumentation code for a JVM. The JVM is built and a program to be tested is then executed on the modified VM. During execution, the runtime system monitors the behavior of the program, checks symptom specifications, and performs amplification and deamplification. PERFBLOWER reports detailed diagnostic information for programs with high virtual overhead.

```
Context ArrayContext {
    sequence = "*.main,*";
    type = "Object[]";
}

Context TrackingContext {
    sequence = "*.main,*";
    path = ArrayContext;
    type = "String";
}

History UseHistory {
    type = "boolean";
    size = UP;//User Parameter
}

Partition P {
    kind = all;
    history = UseHistory;
}
```

```
TObject MyObj{
    include = TrackingContext;
    partition =  P;
    instance boolean useFlag =
        false; //Instance Field
}

Event on_rw(Object o, Field f,
            Word w1, Word w2){
    o.useFlag = true;
    deamplify(o);
}

Event on_reachedOnce(Object o){
    UseHistory h = getHistory(o);
    h.update(o.useFlag);
    if(h.isFull()
        && !h.contains(true)){
         amplify(o);
    }
}
```

Figure 2 An ISL program amplifying memory leaks caused by unnecessary references from arrays.

ISL overview. As the first step, the developer writes an ISL program to describe (1) symptoms of a performance problem (for which amplification is needed) and (2) counter-evidence that shows what is being tracked is not a performance problem (for which deamplification is needed). ISL explicitly models heap partitioning, history of heap updates, as well as collaborations between the collector and the mutator, allowing developers to easily write simple Java-like code to amplify and deamplify performance problems.

To illustrate, Figure 2 shows a sample ISL program that describes the symptom of memory leaks caused by unnecessary strings held in object arrays, as well as how their effects can be amplified. A typical ISL description consists of a set of constructs that specifies how heap updates can be tracked and where the tracking information should be stored, as well as a set of *events* that uses imperative constructs in regular Java to define actions to be taken when the events are triggered.

Context. Many dynamic analyses do not need to track every single detail of the execution. For different performance problems, the developer may focus on different aspects of the execution, such as the behavior of an object when it is created under a certain calling context (e.g., control-related) or referenced by a certain data structure (e.g., data-related). A context construct is designed for the developer to express the scope of the objects of interest. Context has three properties: sequence, type, and path, each of which defines a constraint in a different aspect. They can be used to specify, respectively, the calling context, the type, and the reference path for the objects to be tracked. For example, context `ArrayContext` narrows the scope of the heap to objects of type `Object[]` created under calling contexts that match string “*.main, *”, where * is used as a wildcard. This string matches all call chains that start with an invocation of method “*.main”.

Next, context `ArrayContext` is used to define the reference path in a new context `TrackingContext`, which describes the constraint that we are only interested in the `String` objects that are (1) created under context “*.main, *” (i.e., defined by sequence), and (2) reachable directly from the array objects created under the same calling context in the object graph (i.e., defined by path). Note that this context unifies two orthogonal object naming schemes (i.e., call-chain-based and access-path-based), providing much flexibility for the developer to narrow down interesting objects.

History. Since we target heap-related performance problems, their symptoms can often be identified by comparing old and new values in the tracked objects. A History construct is such an abstraction that models the heap update history (i.e., in an execution window) that needs to be tracked on the objects of interest. A history has two important properties: the type of a history element (i.e., type) and the length of the execution window tracked by the history (i.e., size). The element type has to be a primitive type because tracking a program execution often requires collaborative work between the mutator and the collector; creating tracking objects during a GC would not be allowed. The length of the execution window specifies how many user-defined state updates can be recorded in the history. The recorded updates will be used to determine whether a symptom is seen. A history has to be attached to a partition, allowing for the tracking of heap updates at different heap abstraction levels. We provide a few functions to manipulate a history; these functions will be discussed shortly.

Partition. Different analyses may need to collect tracking information at different abstraction levels. For example, analyses that report information regarding allocation sites can maintain a piece of tracking information for each allocation site, while analyses designed to identify type-state bugs have to keep per-object tracking information. A Partition construct can be used to define the partitioning of the heap needed by an analysis. Properties kind and history specify, respectively, how the heap should be partitioned and what history should be attached to each heap partition. In other words, one history instance (as defined in History) will be created and maintained for each partition of the heap defined by kind. kind can have five different values – all, context[i], alloc, type, and single – which specify heap abstractions with increasing levels of granularity.

In particular, single means that all heap objects will be in the same partition, and therefore will share the same tracking information (i.e., a history instance). type and alloc specify that our runtime system will create one history instance per Java type and per allocation site, respectively; context[i] informs the runtime system to create one history instance for each allocation site executed under a distinct calling context; since each allocation site can have an extremely large number of distinct contexts, we allow the developer to define a (statically-fixed) integer number i to limit the number of contexts for each allocation site; a hash-based encoding function proposed by Bond and McKinley [4] is used to map a full context to a number in [0, i - 1]; this option can be used to enable context-sensitive tracking of certain heap properties; finally, all means that one history instance will be created per heap object.

TObject. TObject defines the type of objects to be tracked, using two properties, include and partition, that specify the tracking context and the heap partitioning, respectively. In our example, the system tracks objects under the context `TrackingContext` and creates a history for each tracked object. In addition, each tracked object has an instance field `useFlag`, which stores object-local state information necessary for identifying the symptom. The keyword `instance` can be used to declare (primitive-typed) per-object metadata information.

Event. At the center of an ISL program is a set of event constructs that define what to do when important events occur on tracked objects. ISL supports seven different events: alloc, read, write, rw, call, reached, and reachedOnce. The first five are regular mutator events while the last two are GC events. reached is triggered every time an object is reached during a GC object graph traversal. Since an object may be reached multiple times (through different references), we use reachedOnce to define actions that need to be taken only once

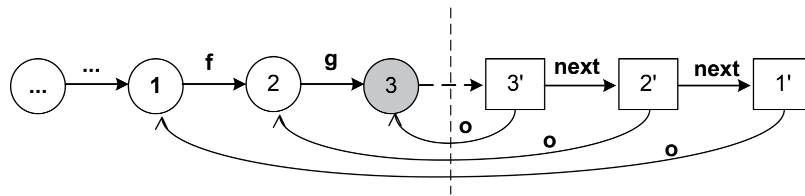


Figure 3 An example mirror path.

on the object during each GC – reachedOnce is triggered only at the first time the object is reached. An event construct takes a few parameters exposing the related run-time values at the event. In Figure 2, the parameters of the `rw` event include the base object, the field being accessed, and the value being read/written. Word is a special type representing a 32-bit value (regardless of its Java type). If this event reads/writes a 64-bit value, this value will be broken into two words w_1 and w_2, representing the high and the low 32 bits, respectively.

In our example, the events `rw` and `reachedOnce` specify the symptom of a memory leak and when to amplify its effect. Once a tracked object o is used (i.e., read or written), we set its `useFlag` true. Since we see a use of the object, we call a library method `deamplify` to cancel all the space penalty previously added to o. When o is reached in a `reachedOnce` event, we update the history associated with o's partition with $o.useFlag$. Since the history is defined to track only the most recent UP[2] updates, the `update` operation will add the current boolean value and discard the oldest value from the history if it is full. Finally, if the history has a full record of UP updates (i.e., `isFull` returns `true`) and the record does not contain any `true` value, object o is stale and thus we add space penalty to amplify the staleness effect by calling method `amplify`.

Virtual amplification and deamplification. Amplification of a performance problem is done on the objects that satisfy the symptom specification of the problem. Method amplify(o) takes an object o as input and increases the PC in o by the size of the object. When deamplify is executed on o, we set o's PC back to zero. During each GC, we modify the reachability analysis to compute the sum of the size of the live heap (S) and the PCs of all live objects (P). We define VSO to be $(P + S)/S$, which simulates the space overhead of an execution had the real memory penalty been used. VSO is a metric that measures the severity of performance problems relative to the amount of memory available to a program – the same problem is more serious if the program is run with a small heap than with a large heap. Since P has a time component (e.g., objects that keep satisfying the symptom specification will make P keep growing), a program may have increasingly large VSOs as it executes. Eventually, the VSOs computed at all GCs are compared and the maximal VSO is reported to indicate the severity of the performance problems.

Providing diagnostic information by mirroring reference paths. When a test fails, it is important to provide highly-relevant diagnostic information that can help developers quickly find the root cause and fix the problem. Evidence [36, 41] shows that heap reference paths leading to suspicious objects are very important information as they reveal calling contexts and data structures containing these objects. However, it can be quite expensive for the runtime system to identify and report reference paths for objects. Although the GC traverses all live heap objects, path information is not easy to obtain – to be scalable,

[2] UP is an acronym for user parameter.

the reachability analysis in the GC uses a breadth-first algorithm that does not record any backward information. In addition, reporting a reference path requires recording the source code information (e.g., allocation site) of each object on the path, which can introduce significant runtime overhead.

In PerfBlower, we solve the problem by building a *mirror object chain* for o that reflects the major reference path leading to o. Figure 3 shows an example mirror path. Each object in the original Java heap is annotated with an integer i and its corresponding mirror object is annotated with i'. To report object 3's reference path, for instance, we only need to traverse *forward* its mirror chain (as apposed to traversing *backward* on the graph) and print information contained in each mirror object. This eliminates the need to either modify the GC algorithm or increase the size of a regular object to store source code information. PerfBlower tracks only one single reference path for each suspicious object. We find it to be sufficient in most cases. A more detailed discussion can be found at the end of Section 4. After the execution finishes, allocation sites are ranked based on the total size of the memory penalties added to their objects; their related reference paths are reported as well.

3 ISL: A Systematic Way to Describe Symptoms

ISL syntax and semantics. In this section, we briefly describe the core parts of the syntax and semantics of ISL. An ISL program is a set of contexts, histories, partitions, tracked objects, and events. The path field of a context c_1 is defined by another context c_2, which specifies that the objects constrained by c_1 have to be directly reachable from the objects constrained by c_2 on the object graph. If we wish to specify transitive reachability, e.g., via a path of n nodes, we can define n contexts from c_2 to c_{n+1}, each constraining a node on the path. History supports the following seven operations: int length(), boolean isFull(), void update(T), T get(int), boolean contains(T), boolean containsSameValue(), and History subHistory(int, int). The first five operations are defined in expected ways. containsSameValue returns true if all elements of the history have the same value; subHistory converts a sub-region of the current history into another history instance.

PerfBlower supports simultaneous checking of multiple problems. This is done by assigning a unique ID to each ISL program specifying a problem. The ID will be passed (implicitly) as a parameter to `amplify`(o) – it is written into each mirror object created for o so that when the mirror path for o is traversed and printed, the description of the problem to which o is related is reported as well.

Example ISL programs. In addition to the memory leak example given in Section 2, Figure 4, Figure 5, and Figure 6 show, respectively, the simplified ISL descriptions for amplifying three other types of performance problems: (a) extensive creation of objects with invariant data contents, (b) under-utilized Java maps, and (c) unused return values. While there are multiple ways to find these problems, we show how a unified amplification-based approach can be used to detect all of them.

Allocation sites creating invariant objects. The symptom is that certain allocation sites keep creating objects with identical data values. The goal of the ISL program is to identify these allocation sites and then penalize their objects. The partitioning is based on allocation site, which means all objects created by the same allocation site share one history instance. We define the history as a list of `long` values (as `long` can express values of any type) over an execution window of UP updates. Here we are interested in values stored in each primitive-

```
History UpdateHistory {
  type = "long";
  size = UP * MAX_NUM_FIELDS;
      //UP means User Parameter
}

TObject MyObj { partition =  P;}

Event on_reachedOnce(Object o){
  UpdateHistory us =
                getHistory(o);
  for(int i = 0;
      i < us.length();i += UP){
    UpdateHistory h =
           us.subHistory(i, UP);
    if(!h.isFull() ||
        !h.containsSameValue())
          return;
  }
  amplify(o); }
```

```
Partition P {
  kind = alloc;
  history = UpdateHistory;
}

Event on_write(Object o,
     Field f, Word w1, Word w2){
  if(f.isPrimitive()){
    long l = combine(w1, w2);
    UpdateHistory hs =
           getHistory(o);
    UpdateHistory h =
           hs.subHistory
             (f.index() * UP, UP);
    h.update(l);
    if(h.isFull() &&
        !h.containsSameValue())
      deamplify(o);
  }
}
```

Figure 4 An ISL description for amplifying allocation sites creating objects with invariant data contents.

```
Context MapContext {
  type = "java.util.Map";
}

Context TrackingContext{
  type = "Object[]";
  path =  MapContext;
}

History UtilityRates {
  type = "double";
  size = UP;//User Parameter
}

Partition P {
  kind = all;
  history = UtilityRates;
}

TObject MyObj{
  include = TrackingContext;
  partition =  P;
}
```

```
Event on_reachedOnce(Object o){
  UtilityRates h = getHistory(o);
  Object[] arr = (Object[]) o;
  int numNonNull = 0;
  for(Object ele : arr){
    if(ele != null)
      numNonNull ++;
  }
  double rate = (double)
          numNonNull/arr.length;
  h.update(rate);
  if(rate > 0.5) deamplify(o);
  else if(h.isFull()){
    for(int i = 0; i < UP; i ++){
      if(h.get(i) > 0.5)
        return;
    }
    amplify(o);
  }
}
```

Figure 5 An ISL description for amplifying under-utilized Java maps.

```
History UseHistory {
  type = "boolean";
  size = UP; //User Parameter
}

Event on_rw(Object o, …){
  if(o.returned){
    getHistory(o).update(true);
    deamplify(o);
  }
}

Event on_reachedOnce(Object o){
  UseHistory h = getHistory(o);
  if(o.returned &&
      h.isFull() &&
      !h.contains(true))
    amplify(o);
}
```

```
Partition P {
  kind = all;
  history = UseHistory;
}

TObject MyObj {
  partition =  P;
  instance boolean returned =
                   false;
}

Event on_call(Object o, Method m,
        Word ret1, Word ret2){
  if(m.getRetType().
             isReference()){
    Object obj = addrToObj(ret1);
    obj.returned = true;
  }
}
```

Figure 6 An ISL description for amplifying never-used return objects.

typed field of an object. Since our History construct can model only scalar values, we *linearize* histories of all fields into one single history whose size is UP $*$MAX_NUM_FIELDS where MAX_NUM_FIELDS is the maximal number of fields in a class (e.g., 100 is a reasonably large number). In other words, for a field with index i, elements from $i*$UP to $(i+1)*$ UP in the history model the updates of the field.

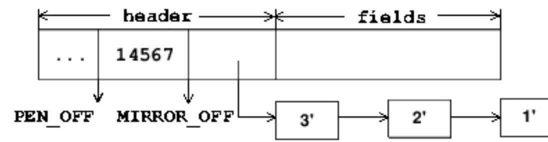
In the write event, we first obtain the sub-history for the field being accessed (f) from the history associated with the object, and then update the sub-history with the value to be written into the field. If a different value is seen in the sub-history, the object is no longer invariant and we cancel all the penalty previously added to the object. In the reachedOnce event, we check whether the sub-history for each field has a full record in which all values are the same. If it is the case, we start penalizing the object. Note that we can make amplification even finer-grained by using a context-based partitioning (e.g., kind = context[i]), which will enable us to identify certain contexts that are more likely to create identical objects than others and only amplify objects created under such contexts.

Under-utilized Java maps. The goal is to penalize Java maps that take a large memory space but contain only a very small number of elements. Using two contexts, we narrow our focus onto object arrays that are referenced by objects of any subtype of `Map`. We use a per-object partitioning with a history that tracks the most recent UP *utility rates* of each array. A utility rate of an array is defined as the ratio between the number of non-null elements and the length of the array. Every time a tracked array is traversed in the GC, we compute its utility rate and update the history. If the rate is greater than a user-provided threshold value (i.e., 0.5), the array is in good health and thus we cancel all its previous penalty. If the history has a full record in which all elements are smaller than 0.5, we start penalizing this array.

Never-used return values. Our goal is to detect and penalize objects that are never used after being returned by a method. Such objects are often indicators of wasteful computation done during the method invocation. Amplifying this problem is similar to amplifying a memory leak. The only difference is that we use a boolean instance field `returned` in each object to indicate whether the object has been returned by a method call. This flag is set in event on_call if a method call returns an object.

Summary. Observe that ISL has the following four advantages. (1) It has a Java-like syntax and thus is easy to use in real-world development; amplification for even complicated performance problems can often be specified using only a few lines of ISL code. On the contrary, had a JVM been modified manually to implement amplification, developing a testing tool for each problem would have needed modification of thousands of lines of code, which can take a skillful programmer several months or even longer.

(2) The `amplify` and `deamplify` commands enable developers to easily develop dynamic analyses with fewer false positives, making PerfBlower less sensitive to user parameters and heuristics. For example, although PerfBlower still needs a user threshold i to determine when to perform amplification, its reliance on finding a perfect i is significantly reduced by turning a symptom into a *cancelable penalty*. In our experiments, we observe that a small i often works very well – if a program only has benign problems, although penalties are still created when a symptom appears, these penalties will be removed later by deamplification and thus not accumulate. As demonstrated in Section 5.4, for large applications such as



(a) The layout of a tracked object

```
Class Mirror{
  Object  o;     // the actual object it mirrors
  AllocSiteInfo info; // source code info of o's alloc site
  Field fInfo;   // information of the corresponding edge
  Mirror next;   // link to the next mirror object
}
```

(b) Metadata Class Mirror

Figure 7 The layout of a tracked object at run time and the metadata class `Mirror` in PERFBLOWER.

`eclipse` and `mysql`, many thousands of false warnings are eliminated by deamplification but would otherwise have been reported by an existing leak detector.

(3) While PERFBLOWER penalizes individual objects, it is easy to attribute penalties to a data structure as a whole by defining reference-path contexts. For example, although array objects are penalized in the detection of under-utilized containers, reference-path contexts are used to connect those low-level arrays with high-level maps, lists, and sets, and hence, PERFBLOWER is able to report not only individual objects but also logical data structures.

(4) By exploiting a combination of instrumentation and runtime system support, ISL provides a unified way of expressing various memory-related performance problems which were defined separately in the existing work.

4 The PERFBLOWER Runtime System

This section describes our amplification runtime system.

ISL compilation. During compilation, contexts are broken into a set of $\langle ms, type\rangle$[3] pairs. A new `History` class is generated from a template based on the properties declared in history. An additional header space is requested per object to accommodate instance fields declared in TObject. Declaring many instance fields would create tremendous space overhead and thus developers are encouraged to declare as few fields as possible. Note that we do not generate a Java class for a TObject construct. The contexts used in the construct are checked at run time to determine whether an event needs to be invoked. We do not typecheck the imperative statements in an event construct; these events are directly translated into Java methods that are invoked at various program points; the generated methods will be typechecked when the RVM code is compiled.

Incrementally adding penalties and mirroring reference paths. Figure 7 (a) shows the layout of a tracked object in PERFBLOWER. We add two words in each object's header space to store (1) its penalty count (in numbers of bytes) and (2) a pointer pointing to the head of its mirror chain. Constants `PEN_OFF` and `MIRROR_OFF` are used to locate the offsets of

[3] *ms* and *type* refer to a method sequence and a type constraint, respectively.

Algorithm 1: Incrementally creating virtual penalties and building mirror paths when AMPLIFY(o) is invoked.

```
    Input: Object obj to be penalized, Object graph G
    Output: Object graph G' with additional mirror objects, and the VSO computed in this GC
 1  Map newRefEdges // Reference edges to be added in the mirror chain
 2  Set objToBeMirrored // Objects to be mirrored
 3  List mirrorObjCurrGC // Mirror objects at the end of each mirror chain
 4  List mirrorObjLastGC // Chain-ending mirror objects found in the previous GC
 5  long VP ← 0 // the total size of the virtual penalty
 6  BEGIN procedure PREGC
 7  lastMark ← currMark // The initial value of currMark is 1
 8  currMark ← currMark%2 + 1
 9  END procedure PREGC
10  BEGIN procedure INGC // that traverses object currObj
11  if currObj = obj /*The object to be penalized is currently being traversed*/ then
12      s ← LOAD(obj, PEN_OFF) + SIZE(obj)
13      STORE(obj, PEN_OFF, s)
14      Mirror head ← LOAD (obj, MIRROR_OFF)
15      if head = null   //The mirror chain does not exist yet then
16          objToBeMirrored ← objToBeMirrored ∪ {obj}
17      else
18          Mirror m ← head
19          Mirror n ← m.next
20          while n ≠ null do
21              if refExists(n.o, m.o) then
22                  m ← n
23                  n ← n.next
24              else
25                  // the recorded path has changed
26                  m.next ← null
27                  break
28          // mark the object n mirrors
29          SETMARK(n.o, currMark)
30          mirrorObjCurrGC ← mirrorObjCurrGC ∪ {n}
31  foreach object p referenced by currObj do
32      if GETMARK(p) = lastMark then
33          newRefEdges ← newRefEdges ∪ {⟨currObj, p⟩}
34          UNMARK(p)
35  VP ← VP + LOAD (obj, PEN_OFF)
36  END procedure INGC
37  BEGIN procedure POSTGC
38  foreach Object m ∈ objToBeMirrored do
39      Mirror m' ← CREATEMIRROR()
40      WRITESOURCECODEINFO(m, m')
41      STORE(m, MIRROR_OFF, m')
42  foreach Mirror object p ∈ mirrorObjLastGC do
43      if ∃⟨i, j⟩ ∈ newRefEdges: j = p.o then
44          /*The edge between i and j needs to be mirrored*/
45          Mirror i' ← CREATEMIRROR()
46          WRITESOURCECODEINFO(i, i')
47          p.next ← i'
48  mirrorObjLastGC ← mirrorObjCurrGC
49  mirrorObjCurrGC, objToBeMirrored, newRefEdges ← ∅
50  VSO ← (VP + LIVESPACESIZE())/ LIVESPACESIZE()    //compute virtual space overhead
51  END procedure POSTGC
```

these two locations upon accesses. Figure 7 (b) shows the definition of the `Mirror` class. Each mirror object contains two outgoing edges, one pointing to the next mirror object (i.e., via field `next`) and the other pointing to the actual object it mirrors (i.e., via field *o*). The mirror object also contains the source code information of the object it mirrors as well as the field information of the corresponding reference edge in the original path. In the example mirror chain shown in Figure 3, field `fInfo` in object 3' and object 2' record field *g* and *f* respectively.

Algorithm 1 shows our algorithm for incrementally adding penalties and building mirror paths. The algorithm has three major procedures: PREGC, INGC, and POSTGC. PREGC sets two mark values that will be used later in GC to mark objects (lines 6–9): *lastMark* and *currMark*, which represent, respectively, the mark values used in the last and the current GC, alternate between 1 and 2. Procedure INGC traverses the object graph to build mirror reference paths (lines 10–36). Because the object graph traversal cannot go backward, it is impossible to mirror a reference path with multiple edges in one single GC run. The basic idea of the incremental algorithm is *to mirror one edge on the path in each GC, starting from the one closest to the penalized object.* For example, to build the mirror path in Figure 3, we first mirror the edge going directly to object 3 (i.e., $2 \xrightarrow{g} 3$) by creating an edge between $3'$ and $2'$. Next, we mark object 2 with a special mark value so that 2 will be recognized in the next GC and the edge between 1 an 2 can be mirrored. If object 3 keeps being penalized (e.g., indicating that we are amplifying a true problem), a long mirror chain will be constructed and, hence, more complete path information will be reported.

During the graph traversal, if the object being reached happens to be the one to be penalized (lines 11–30), we first increase the object's PC by the size of object (lines 12 and 13), and then find the head object of its mirror path (line 14). If no mirror object has been created (i.e., it is the first time to penalize this object), we remember the object in set *objToBeMirrored* – since we cannot create mirror objects during a GC, we will do it later after the GC. If the head mirror object is found, we need to create a new mirror object and append it to the existing mirror path. Since this existing mirror path was built in the previous GCs, certain reference edges being mirrored may have changed. To detect such changes, we traverse the mirror path (lines 18–27) to validate if each edge in the path still mirrors a valid heap reference (line 21). When the traversal finishes, *n* is either the last node of the mirror path or the first node at which the old reference relationship breaks. A new mirror object will be created and linked to *n*. We mark the object that *n* mirrors (i.e., *n.o*) with *currMark* (line 29) so that *n.o* will be recognized in the next GC and a new reference edge pointing to *n.o* will be mirrored. *n* is then remembered in set *mirrorObjCurrGC* and we leave the mirror object creation to the PostGC procedure.

If any child of the object being traversed has been marked (by the previous GC) (line 32), the reference edge connecting the object and this child needs to be mirrored. We remember the pair $\langle currObj, p\rangle$ in map *newRefEdges* for further processing (line 33) and then unmark object *p* (line 34). It is important to note that the correctness of an existing mirror chain is verified when its root node is traversed during each GC (lines 25–27). If the reference path it mirrors becomes invalid, the mirror chain will be removed from the root node and the system starts mirroring the new reference path.

All mirror objects are created after the GC is done (lines 38–47). We first create a mirror object m' for each object $m \in objToBeMirrored$ that does not have a mirror chain, store *m*'s allocation site information into m' (line 40), and write a reference of m' into *m*'s header space (line 41). Next, we check each chain-ending mirror object $p \in mirrorObjLastGC$ and see whether there exists a new heap reference edge recorded in *newRefEdges* whose target is

mirrored by p (line 43). If such an edge exists, we create one more mirror object to mirror the source of the edge (lines 45–47).

All objects in list *mirrorObjCurrGC* are added to list *mirrorObjLastGC* before the mutator execution resumes, so that these objects will be used to mirror new edges in the next collection. Finally, the data structures *mirrorObjCurrGC*, *objToBeMirrored* and *newRefEdges* are cleared (line 49), and the VSO is computed (line 50) based on the penalty size VP (which is updated at each node traversal at line 35).

Our algorithm can mirror only one reference path for each object. We find it to be sufficient in most cases: there is often one single reference path in which an object is inappropriately used and causes a problem, although the object may be referenced by many paths at various points; if the object keeps being penalized, the penalty chain will eventually get stabilized to mirror the problematic path necessary for the developer to fix the problem. Furthermore, for many types of performance problems, reporting *any* reference path will be helpful to find their root causes. For instance, for leak detection, any reference path that holds a suspicious object and makes it reachable is problematic. As another example, for detection of under-utilized containers, PerfBlower tracks inefficiently-used arrays and the key to producing a high-quality report is to find the data structures (e.g., HashMap, ArrayList, etc.) that reference those arrays. Since an array is often *owned* by a high-level data structure, any reference path that leads to the array must pass the data structure, and hence, reporting any path will be sufficient for the developer to find the data structure and understand the problem.

Limitations. While PerfBlower captures commonalities of different memory-related problems and makes it easy for them to manifest, it has a few limitations that leave room for improvement. First, it can only find heap-related inefficiencies, while there are many different sources for real-world performance problems, such as idle threads, redundant computations, or inappropriate algorithms. Second, the JVM needs to be rebuilt every time a new checker is added. In practice, this is not an issue, since most modern JVMs support fast and incremental building. For example, it takes only one minute to build the Jikes RVM on which PerfBlower is implemented. One future direction would be to make PerfBlower a JVMTI-like library that can be dynamically registered during bootstrapping of the JVM without needing to build the JVM. Third, since PerfBlower piggybacks on the GC, its effectiveness may be affected by the GC frequency. PerfBlower may report a higher overhead if a program is executed with a smaller heap (due to more GCs). While this introduces uncertainties, our experimental results (in Figure 8) demonstrate, for most programs, the VSOs reported by PerfBlower are very stable across different heap sizes. Finally, PerfBlower may become less effective when a generational GC is used. Since a generational GC does not frequently perform full-heap scanning, objects in the old generation space may not be effectively amplified.

5 Evaluation

To implement the proposed technique, we have modified both the baseline compiler and the optimizing compiler in the Jikes Research Virtual Machine (RVM), version 3.1.3, which uses the MarkSweep GC. PerfBlower is now available on BitBucket for download. Using ISL, we have implemented three different amplifiers that target, respectively, memory leaks, under-utilized containers, and over-populated containers. In our evaluation, no application-related information was added into these descriptions, although a future user may describe a

Table 1 Virtual space overheads (in times) reported by the amplifiers for memory leaks (section (a)), under-utilized containers (section (b)) and over-populated containers (section (c)); each section reports the real space overheads (*Inf*) and the VSOs when the size of the history m is 10. Highlighted in the table are the programs whose maximal VSO is greater than 15. Notation of the form $a \rightarrow b$ means a program with a VSO a has a new VSO b after its performance problems are fixed. GeoMean-H and GeoMean-L report the average VSOs for the highlighted and non-highlighted programs, respectively.

Bench	(a) MEM		(a) UUC		(b) OPC	
	Inf	$m = 10$	*Inf*	$m = 10$	*Inf*	$m = 10$
antlr	1.20	1.1	1.22	2.6	1.20	1.0
bloat	1.35	1.1	1.72	5.9	1.58	1.1
eclipse	1.28	5.6	1.21	8.7	1.23	3.5
fop	1.15	1.6	1.22	3.3	1.16	1.1
hsqldb	1.24	**26.2**→1.3	1.19	**16.6**→2.2	1.19	**29.2**→1.7
jython	1.16	**22.7**→6.4	1.06	**48.3**→5.3	1.06	**16.3**→3.8
luindex	1.18	1.2	1.16	1.7	1.13	1.1
lusearch	1.60	1.4	1.70	1.5	1.69	1.2
pmd	1.18	1.5	1.12	1.4	1.12	1.1
xalan	1.15	**53.1**→7.1	1.09	**117.7**→7.9	1.08	3.4
GeoMean-H		**31.6**		**45.5**		**21.8**
GeoMean-L		1.6		2.9		1.5

symptom in a way that is specific to an application. These ISL programs were easy to write and took us only a few hours. All experiments were executed on a quadcore machine with an Intel Xeon E5620 2.40GHZ processor, running Linux 2.6.32.

5.1 Amplification Effectiveness

The first set of experiments focuses on understanding of whether our technique is effective in exposing performance problems in a testing environment. Since the DaCapo benchmark set [2] provides three different kinds of workloads (i.e., small, default, and large), we ran each program on its small workload to simulate how a developer would use our tool during testing – it is much more difficult to reveal performance bugs under small workloads than large workloads. We make no assumption about test inputs – real-world developers can enable amplification when executing a regular test suite; whenever a test case triggers a true problem, amplification will incur a large virtual overhead.

Indication of performance bugs. Table 1 reports the real and virtual space overheads of the three amplifiers over the DaCapo benchmarks (2006 release) using a 500MB heap. The comparison of the amplification results under different heap sizes will be discussed shortly. We discuss our experimental data only for $m = 10$, where m is the user-specified history length. Other configurations have yielded similar results; detailed results for the memory leak detector are reported in Table 2. Clearly, these VSOs indicate that leaks may exist in `hsqldb`, `jython` and `xalan`; under-utilized containers in `hsqldb`, `jython` and `xalan`; and over-populated containers in `hsqldb` and `jython`.

An important observation made here is that there is a clear distinction between the overheads of programs with and without problems, making it particularly easy for developers to determine whether manual inspection is needed during testing. For example, to find over-populated containers, there are only two programs whose memory consumption is significantly amplified. Developers only need to read reports for these two programs; they

Table 2 Virtual space overheads (in times) of the memory leak amplifier in different configurations using a 500MB heap; section m=i shows, for each program, the maximal VSO when the length of the execution window (i.e., field `size` of a history) is i. Highlighted in the table are the programs whose maximal VSO is greater than 15. Notation of the form $a \to b$ means a program with a VSO a has a new VSO b after its performance problems are fixed. GeoMean-H and GeoMean-L for a section $m = i$ report the average VSOs under $m = i$ for the highlighted and non-highlighted programs, respectively.

Bench	$m = 5$	$m = 10$	$m = 15$	$m = 20$
antlr	1.2	1.1	1.2	1.2
bloat	1.1	1.1	1.1	1.1
eclipse	5.3	5.6	6.3	6.3
fop	1.6	1.6	1.7	1.7
hsqldb	**26.5**→1.7	**26.2**→1.3	**25.9**→1.2	**25.7**→1.2
jython	**22.7**→6.3	**22.7**→6.4	**22.7**→6.4	**24.5**→6.3
luindex	1.2	1.2	1.2	1.2
lusearch	1.4	1.3	1.3	1.3
pmd	1.5	1.5	1.5	1.5
xalan	**53.1**→7.3	**53.2**→7.1	**40.4**→2.7	**37.0**→1.9
mysql	**155.5**→2.3	**108.9**→2.3	**70.4**→1.5	**30.1**→1.2
mckoi	**111.6**→6.9	**72.4**→6.6	**85.6**→3.3	**42.6**→1.7
GeoMean-H	**56.1**	**47.8**	**42.7**	**31.3**
GeoMean-L	1.6	1.6	1.6	1.6

would otherwise have to inspect all programs and check each warning had a regular detector been used. The differences in the VSOs of programs with and without problems are clearly demonstrated by the numbers reported in the GeoMean-H and GeoMean-L rows. We have manually inspected the programs whose VSO is > 15[4] and found problems in all of them. Details of these problems will be discussed shortly in Section 5.2. Note that after their performance problems are fixed, the VSOs of these problems are significantly reduced.

Space and time overheads. Section *Inf* of Table 1 reports the real space overheads of our infrastructure, computed as S_1/S_0, where S_0 and S_1 are, respectively, the peak post-GC heap consumptions of the unmodified and modified RVM. The average space overheads incurred by the three amplifiers (i.e., memory leaks, under-utilized containers, and over-populated containers) under the configuration $m = 10$ are 1.23×, 1.23×, and 1.25×, respectively. The overall time overheads under the same configuration are 2.39×, 2.74×, and 2.73×, respectively. To understand the scalability of the tool, we have also run these amplifiers using the DaCapo large workloads. The time overheads for the three amplifiers are all between 2.5× and 3.2×.

Sensitivity to the execution window length. We ran the memory leak amplifier with $m = 5$, $m = 10$, $m = 15$ and $m = 20$ to show whether our framework is sensitive to the execution window length. Table 2 shows the results for the DaCapo benchmarks with different m. We added two extra programs `mysql` and `mckoi` to our benchmark set – they contain known leaks studied before and we are thus interested in how our amplifier performs on these programs.

We make two observations based on the amplification results. First, the VSOs under different parameters are generally stable; the size of an execution window does not have much impact on the amplification results. On the contrary, the effectiveness of an existing

[4] 15 is just a number in a large range (between 10 and 20) that can easily distinguish these overheads.

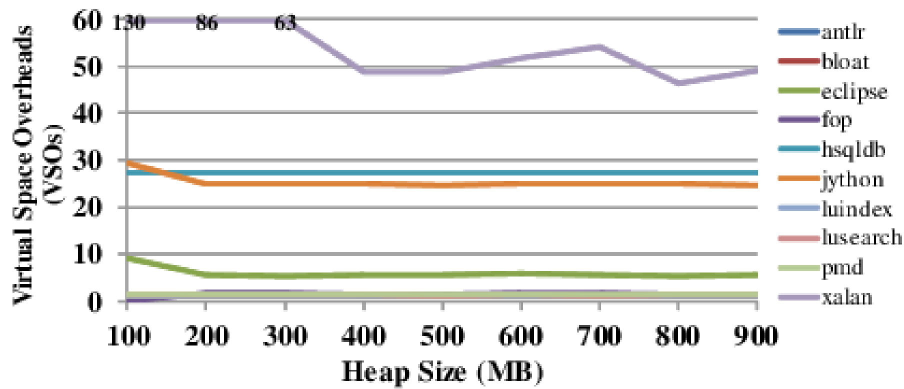


Figure 8 VSOs for the memory leak amplifier under different heap sizes.

symptom-based performance problem detector relies heavily on such a threshold. Second, as the size of the execution window increases, the space overhead of the tool decreases in general. This is straightforward – the larger the size of the history is, the fewer objects are amplified, and the smaller effect amplification creates.

Sensitivity to the heap size. Are VSOs sensitive to the size of the heap? As demonstrated in Figure 8, the VSOs reported by PERFBLOWER are quite stable under different heap sizes.

5.2 Problems Found

We have manually inspected our reference path report for each highlighted program in Table 1 and Table 2. We have not found any highlighted program to be mistakenly amplified, that is, real performance problems are identified in all of them. We have studied a total of 10 amplification reports and found 8 unknown problems that have never been reported in any prior work. Furthermore, all of these problems are uncovered under small workloads; their executions do not exhibit any abnormal behavior without amplification. In this section, we discuss our experiences with 8 unknown problems. To reduce the influence of execution noise, the performance gains shown below are obtained by running each modified program twenty times and comparing the medium running time and space consumption with those of its unmodified version. For each problem found, we have developed a fix. As shown in Table 1 and Table 2, the VSOs of these programs are significantly reduced after their fixes are applied. This demonstrates that PERFBLOWER is sensitive only to true memory issues, not false problems.

For each program, we have also run an existing leak detector, SLEIGH [3], and compared our reports with SLEIGH's reports. The reason we chose SLEIGH is because it is the only publicly available performance problem detector that does not require user involvement. While there exist other tools such as LeakChaser [36] and a few inefficient data structure detectors [26], they are either unavailable or require heavy user annotations to understand the program semantics. Hence, we compare PERFBLOWER only with SLEIGH in this paper.

xalan-leak. The No. 1 allocation site in the report has a very long reference path (with 10 nodes); the top 3 objects on the path are as follows:

```
#0. id: 5561 ObjectVector.java: 106
Method: org.apache.xml.utils.ObjectVector.<init>
Object Type: java.lang.Object[]
Number of suspicious objects: 27
Size: 0x00d84380
Path: ID: 5573 XPathContext.java: 920
   Method: org.apache.xpath.XPathContext.<init>
   Object Type: org.apache.xml.utils.ObjectStack
   <= ID: 5557 TransformerImpl.java: 402
      Method: org.apache.xalan.transformer.TransformerImpl.<init>
      Object Type: org.apache.xpath.XPathContext
      <= ID: 5543 StylesheetRoot.java: 212
         Method: ...templates.StylesheetRoot.newTransformer
         Object Type: org.apache.xalan.transformer.TransformerImpl
         <= ...
```

The reference path shows us that the largest penalty is added to objects created and referenced by the instances of an XML transformer class `TransformerImpl`. This immediately caught our attention, since `TransformerImpl` objects should be reclaimed after each transformation. The reference path directed us to class `StyleSheetRoot`, which implements interface `Templates` and has a reference to an `IteratorPool` object. Each `IteratorPool` object maintains a vector of `DTMAxisTraverser` objects in order to reuse those objects. However, upon the reuse of an old `DTMAxisTraverser` object, the object still keeps a reference of its previous `SAX2DTM` object, which, in turn, references the `TransformerImpl` object. Hence, each reused `DTMAxisTraverser` object in the `IteratorPool` leaks its previous `TransformerImpl`. This is a serious problem, because the size of a `TransformerImpl` object is often very large (e.g., at the megabyte level). We came up with a quick fix in which one line of code is commented out to disallow reuse of `DTMAxisTraverser` objects. **The fix has resulted in a 25.4% reduction in memory consumption and a 14.6% reduction in execution time.** A subsequent search in the `xalan` bug repository reveals that the same bug was found and fixed in version 2.5.1 [1] with a different approach, while the DaCapo xalan version in which we detected the bug is a much earlier one (2.4.1). We found that every object on this long path is necessary for understanding the cause of the bug. It would not be possible to record such complete information with a regular detector/profiler, as it uses a fixed-size space to store metadata for each object.

For `xalan`, Sleigh reported a few stale objects, including their types and numbers of instances. Most of these types are Java library classes or VM classes, such as `java.lang.String` and `com.ibm.JikesRVM.classloader.VM_Atom`, which have nothing to do with the leak. In this case, Sleigh provided neither allocation sites nor last access sites for these stale objects. This is primarily because Sleigh uses only one bit per object to encode information statistically; it needs a long execution to gather sufficient data for producing a diagnostic report, and thus does not work well for test executions, which are often very short.

jython-leak. From the report, it is easy to identify that most space penalties are added to stale `PyJavaPackage` objects. One use of these objects is to cache the relationships between Java classes and Jar files. When a jar file is loaded, Jython iterates all the classes and creates PyJavaPackage objects. Many classes are never used during execution, and thus, their corresponding `PyJavaPackage` objects become leaks. We fixed the problem by (1) employing lazy loading and (2) removing unnecessary jars from classpath. **The fix has led to a 24.3% peak memory reduction and a 7.4% time reduction for the warm up phase.**

In SLEIGH's report, there was only one stale allocation site. Although this allocation site is related to the cause of the leak, it is still far away from the cause and it would take a developer a fair amount of effort to find the cause from the allocation site. On the contrary, PERFBLOWER provided much more precise diagnostic information, significantly alleviating the developer's debugging burden.

hsqldb-leak. Most of the reported objects are related to data values computed but not used at all in the DaCapo execution. For example, `hsqldb` maintains three maps between paths and databases. Although in DaCapo only the memory database is used, `hsqldb` also keeps maps for the file database and the resource database. One way to solve the problem is to provide configuration options that allow users to specify the databases needed. We changed these stale objects to lazy initialization, reducing the memory consumption by **15.6%**.

hsqldb-UUC. The report shows most under-utilized containers are due to inappropriate initial capacity. For example, the top object in our report points to arrays created during the initialization of `BaseHashMap`. The reference path quickly led us to inspect ValuePool.java. In this class, `hsqldb` caches six `ValuePoolHashMap` objects in static fields, which are used to support singleton patterns. However, all of these maps have 10000 as their initial capacity, which is way too large for many clients. We fixed the problem by making these containers grow dynamically as necessary and changing their initial capacity to 32. **This optimization has led to a 17.4% space reduction.**

jython-UUC. Our report indicates that the largest penalty is added to arrays created in `PyStringMap.resize` method. By inspecting `PyStringMap`, we found that the class uses a string array to store keys and a `PyObject` array to store values. Before each insertion, it checks the load factor of its key array (i.e., how full the map is allowed to get before its capacity is increased). In this case, if the load rate is higher than 0.5, its capacity is doubled. Actually, since the hash function of `String` works generally well and the key array has a very small collision rate, using 0.5 as load factor seems too aggressive. Modifying it to 0.75 resulted in a **19.1% space reduction**.

xalan-UUC. The top warning in the report points to `ObjectStack` objects created in `XPathContext` and `TransformerImpl` objects. The initial capacity of `ObjectStack` is defined by `XPathContext`. `RECURSIONLIMIT`, a static final field with a value 4096. A detailed inspection of the source code reveals that ObjectStack is implemented based on ObjectVector, which is a data structure designed to support random accesses. The initial capacity defines a tradeoff between space and time: using a smaller value saves space at the cost of increased lookup computation. Finding the sweet spot requires deep understandings of the program and empirical studies. When we use 2048 as the initial capacity, the memory consumption is reduced by **5.4%**, and the execution time is reduced by **34.1%**.

hsqldb-OPC. Before `hsqldb` executes an `insert` query, it invokes a method called `getNewRowData` to create an object array that represents the row to be inserted. A column in a database table often has a default value – when a new row specified by an `insert` query does not have a value for the column, this default value will be filled in. We find that method `getNewRowData` creates a large number of objects and fill them into the object array as default values. These objects are never retrieved from this array until later they

Table 3 Comparison of results from PerfBlower and Sleigh; leaks^{+} means the tool reports both memory leaks and useful diagnostic information; leaks^{-} means the tool only reports leaks without useful information.

Bench	PerfBlower	Sleigh
antlr	no leak	no leak
bloat	no leak	no leak
eclipse	leaks^{+}	no leak
fop	no leak	no leak
hsqldb	leaks^{+}	leaks^{+}
jython	leaks^{+}	leaks^{-}
luindex	no leak	false positive
lusearch	no leak	fail to run
pmd	no leak	no leak
xalan	leaks^{+}	leaks^{-}
mysql	leaks^{+}	fail to run
mckoi	leaks^{+}	leaks^{-}
jbb	leaks^{+}	leaks^{+}
true unknown leaks	**4**	2
true known leaks	**3**	**3**
false positives	**0**	1
useful information	**7**	2

are replaced by the actual values. We fix the problem by performing a lazy default value assignment, resulting in a **14.9% space reduction.**

jython-OPC. The cause of the problem here is the same as in jython-leak. The top object in our report directed us to inspect method `PyJavaPackage.resize`, in which a large array is created to contain Java classes, most of which are never retrieved from the map. This same problem has two different manifestations.

Comparison with Sleigh. Table 3 summarizes the comparison of leak reports between PerfBlower and Sleigh. PerfBlower found memory leaks in seven benchmarks (including four new leaks and three known leaks), and for each memory leak, PerfBlower provided very precise diagnostic information. Sleigh reported that six programs contain memory leaks, including three known leaks, two new leaks, and a false warning. Furthermore, Sleigh did not provide any useful diagnostic information for three leaking programs. The comparison demonstrates that PerfBlower is able to find more leaks, has fewer false positives, and generates more precise diagnostic information.

5.3 PerfBlower Completeness

Our amplifiers are able to find bugs in all of the highlighted programs, but do they miss bugs? To answer this question, we performed an additional experiment. In this experiment, the benchmark set includes 14 programs (shown in Table 4), which contain known performance problems reported in the literature [14, 3, 5, 36, 39] except Chameleon [26]. Chameleon is a dynamic technique that can detect inefficiently-used containers. However, it is based on the commercial J9 VM and not publicly available. In addition, the description of the bugs

Table 4 Completeness of PerfBlower: for each existing Java-based performance problem detector, the table reports the bugs studied in its original paper and whether they are found by PerfBlower.

Work	*Bugs studied*	PerfBlower
Cork [14]	SPECjbb's leak	Reported
	Eclipse #115789	Reported
Sleigh [3]	SPECjbb's leak	Reported
	Eclipse #115789	Reported
Container Profiler [38]	SPECjbb's leak	Reported
	JDK #6209673	Cannot execute
	JDK #6559589	Cannot execute
LeakChaser [36]	SPECjbb's leak	Reported
	Eclipse #115789	Reported
	Eclipse #155889	Reported
	MySQL's leak	Reported
	Mckoi's leak	Reported
Leak Pruning [5]	SPECjbb's leak	Reported
	Eclipse #115789	Reported
	Eclipse #155889	Reported
	MySQL's leak	Reported
	JbbMod's leak	Reported
	Mckoi's leak	Reported
	Delaunay's leak	Reported
	List leak	Reported
	Swap leak	Reported
	Dual leak	Reported
Static Bloat Finder[39]	Bloat's UUC	Not Triggered
	Chart's OPC	Cannot execute

in [26] is at a very high level and does not contain detailed location information (such as classes, methods, and line numbers). Hence, those bugs were not considered.

Among these benchmarks, three programs could not be executed. Two of them are about memory leaks in Sun JDK library, which is not used by Jikes RVM. Another benchmark is `chart`, which could not be executed due to the missing of certain AWT libraries. For the remaining bugs, all but one were captured by PerfBlower. The missing one is an under-utilized container in the DaCapo benchmark `bloat`, reported by a static analysis-based container problem detector [39]. Because PerfBlower is based on dynamic analysis, its ability of finding bugs depends on the coverage of test cases. We inspected the source code of benchmark `bloat` and concluded that this bug could not be triggered by the DaCapo input. Overall, PerfBlower did not miss any bug that could be triggered by a test case.

5.4 False Positive Elimination

In order to understand if our technique successfully eliminates false warnings of a regular performance problem detector, we measured the number of objects that have experienced both amplification and deamplification. These objects are amplified because they satisfy the symptom specification for a sufficiently long period; they are deamplified later when the symptom disappears. Although they are not true causes of performance problems, a symptom-based detector would report all of them.

Table 5 False warnings that are eliminated by deamplification but would have been reported by a memory leak detector; for each program under each history size i, we report four numbers *FO*, *FS*, *TO*, and *TS*; *FO* is the number of such objects that their staleness exceeds i but they are used afterwards, *FS* is the number of allocation sites creating objects in *FO*, *TO* is the total number of objects whose staleness exceeds i, and *TS* is the number of allocation sites creating objects in *TO*. Although objects under *FO* are not true leaks, a regular leak detector would report all of them.

Bench	$m = 5$		$m = 10$		$m = 15$		$m = 20$	
	FO/FS	*TO/TS*	*FO/FS*	*TO/TS*	*FO/FS*	*TO/TS*	*FO/FS*	*TO/TS*
antlr	26/2	103/68	10/2	87/68	3/2	80/68	2/1	79/68
bloat	19/3	72/29	4/3	56/29	3/3	54/29	2/2	53/29
eclipse	5992/498	12062/1279	4697/246	10572/1030	4648/235	9747/1018	4339/201	8852/958
fop	27/2	823/723	9/1	806/722	4/1	797/722	0/0	793/722
hsqldb	386/4	1866/105	188/4	1236/105	68/3	793/105	13/3	493/105
jython	28/3	17864/1072	21/3	17709/1034	11/2	17634/951	7/2	17410/830
luindex	9/2	94/72	5/2	90/72	2/2	90/72	2/2	88/72
lusearch	183/15	1713/113	32/14	206/67	5/4	178/66	3/2	174/64
pmd	28/2	238/156	6/2	210/156	1/1	204/156	1/1	203/156
xalan	120/12	19795/637	65/11	14657/637	17/8	11304/636	10/7	10540/628
mysql	4457/3	54937/192	1933/3	40844/192	2/2	26547/192	2/2	12877/192
mckoi	116/76	12004/298	116/76	11547/292	111/76	10133/292	1/1	5058/236
jbb	8192/55	54358/59	27/8	49018/55	5/4	27913/55	4/4	25081/53

We focus only on memory leak amplification in this subsection; our results are shown in Table 5. The setup for this experiment is exactly the same as the one for the experiment reported in Table 2: the DaCapo small workload is used and the number of iterations for each program is such that the execution can have at least 20 GCs. *FO* and *FS* are, respectively, the numbers of falsely amplified objects and their allocation sites; these false warnings are eliminated by amplification but would have otherwise been reported by a regular memory leak detector.

We make three observations based on these numbers. First, the number of false warnings is reduced as we increase the threshold i. This is straightforward because true leaking objects often have larger staleness than false leaks. However, increasing i can easily lead to *false negatives* because the staleness of objects created late during execution may not reach i before the execution finishes. This can be observed from the fact that the total numbers of suspicious objects *TO* and suspicious allocation sites *TS* drop significantly, especially for `lusearch`, `xalan`, `mysql`, and `mckoi`. Our algorithm frees developers from choosing a perfect threshold i: one can always use a small i (such as 5) to ensure no true problem is missing without worrying about false positives – most of them are automatically eliminated by deamplification. Finally, PerfBlower has eliminated not only false leaking objects (*FO*), but also false leaking allocation sites (*FS*). Since a leak detector often ranks and reports allocation sites, elimination of false leaking allocation sites leads directly to more precise reports as well as reduced manual inspection effort.

6 Related Work

Finding performance bugs. Jin et al. [13] study a number of performance bugs in real-world programs and develop a pattern-based approach to find bugs in the program source code. Song and Lu [28] propose a statistical approach to finding performance bugs in real-world software. Xu et al. propose static [39, 40] and dynamic [37] techniques to detect various kinds of performance problems. Recent work from [23] proposes a static analysis that can find redundant container traversals. Mitchell et al. propose heap analysis techniques [20, 19] that can correlate system metrics with program behaviors. Nistor et al. [22] propose a runtime technique to detect performance problems by looking for code loops that exhibit similar memory-access patterns. Han et al. [12] identify performance problems by mining large

numbers of stack traces. Stewart et al. propose EntomoModel [29], a dynamic framework that uses decision tree classification and a design-driven performance model to identify and avoid performance anomaly. Caramel [21] detects loop-related performance bugs that can be easily fixed by adding conditional breaks. These existing works are postmortem debugging techniques, focused on finding the root cause of a performance problem that already manifests in a user-provided test execution or actual production run, while our work provides a general framework to describe and amplify performance problems during testing.

Xiao et al. [34] develop a delta inference technique that predicts workload-dependence performance bugs by using models with respect to workload size to infer iteration counts. Yu et al. [42] propose a technique that collects large numbers of traces to measure performance impacts and identify their root causes. WuKong [44] is an automated technique that builds a feature-based behavioral model to predict bugs that manifest at large-system scale. WuKong falls into the category of *learning-based bug detection* where small-scale training runs are employed to build the model, while our approach uses developer-provided symptom specifications to amplify performance problems.

Test amplification techniques. Test amplification is a notion wherein a single test execution can be used to learn much more information about a program's behavior than is directly exhibited by that particular execution. Zhang et al. [43] proposes a test amplification technique that exhaustively explores the space of exceptional behaviors to validate exception handling code. Work from [16] uses a static information flow analysis to amplify the result of a single test execution over the set of all inputs and interleavings for verifying properties of data parallel GPU programs. Dynamic test generation techniques such as [7, 9, 25] can also be considered as test amplification techniques which generate many tests from one test execution to achieve path coverage.

Domain-specific languages. The Broadway compiler project [11] contains an annotation language and a compiler that can customize a library implementation for specific application requirements. Other domain-specific languages and compilers attempt to either incorporate application semantics into compilation to improve performance (such as [8]) or generate application code from declarative specifications (such as [27]). Annotations such as DyC [10] have been used to direct dynamic optimization. Vandevoorde [32] proposes specifications based on Larch [33] to enable performance optimizations. Work closest to the proposed ISL language is [24] that develops a declarative language called DEAL to describe assertions checkable within one single GC run. Unlike DEAL that aims to inform the GC how to check assertions, ISL is an event-based instrumentation language that specifies (1) how a program should be instrumented, (2) what data need to be collected, and (3) how the mutator and the GC should collaborate.

7 Conclusions

This paper presents the first technique that can expose a class of performance problems during testing by amplifying their effects. We first design a language to describe the symptom of a performance problem if the symptom can be expressed by logical statements over a history of heap state updates. Next, we develop compiler and runtime system support that compiles an ISL program, performs the instrumentation, and amplifies the target problem when the symptom is observed at run time. We have implemented this technique on Jikes RVM and used it to successfully amplify three types of heap data-related performance problems.

Acknowledgements. We would like to thank Michael Bond, Brian Demsky, David Liu, Shan Lu, and Feng Qin for their helpful comments on an early draft of the paper. We also thank the anonymous reviewers for their valuable and thorough comments. This material is based upon work supported by the US National Science Foundation under grant CNS-1321179 and CCF-1409829, and by the Office of Naval Research under grant N00014-14-1-0549.

References

1 Apache JIRA issue tracker. `https://issues.apache.org/jira/browse/XALANJ-796`.
2 S. M. Blackburn and *et al.* The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.
3 Michael D. Bond and Kathryn S. McKinley. Bell: Bit-encoding online memory leak detection. In *ASPLOS*, pages 61–72, 2006.
4 Michael D. Bond and Kathryn S. McKinley. Probabilistic calling context. In *OOPSLA*, pages 97–112, 2007.
5 Michael D. Bond and Kathryn S. McKinley. Leak pruning. In *ASPLOS*, pages 277–288, 2009.
6 Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley, 2010.
7 Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *CCS*, pages 322–335, 2006.
8 Dawson R. Engler. Incorporating application semantics and control into compilation. In *DSL*, pages 9–9, 1997.
9 Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.
10 Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theor. Comput. Sci.*, 248(1-2):147–199, October 2000.
11 Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *DSL*, pages 39–52, 1999.
12 Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE*, pages 145–155, 2012.
13 Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88, 2012.
14 Maria Jump and Kathryn S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *POPL*, pages 31–38, 2007.
15 JUnitPerf. `http://www.clarkware.com/software/JUnitPerf.html`, 2003.
16 Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. Verifying GPU kernels by test amplification. In *PLDI*, pages 383–394, 2012.
17 Load Test Tools. `http://www.softwareqatest.com/qatweb1.html`, 2015.
18 Microbenchmarking framework for Java. `https://code.google.com/p/caliper/`, 2013.
19 Nick Mitchell and Gary Sevitsky. The causes of bloat, the limits of health. In *OOPSLA*, pages 245–260, 2007.
20 Nick Mitchell, Gary Sevitsky, and Harini Srinivasan. Modeling runtime behavior in framework-based applications. In *ECOOP*, pages 429–451, 2006.
21 Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *ICSE*, 2015.
22 Adrian Nistor, Lintao Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *ICSE*, pages 562–571, 2013.

23 Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *PLDI*, 2015.

24 Christoph Reichenbach, Neil Immerman, Yannis Smaragdakis, Edward Aftandilian, and Samuel Z. Guyer. What can the GC compute efficiently? A language for heap assertions at GC time. In *OOPSLA*, pages 256–269, 2010.

25 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *FSE*, pages 263–272, 2005.

26 Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: Adaptive selection of collections. In *PLDI*, pages 408–418, 2009.

27 Yannis Smaragdakis and Don Batory. DiSTiL: a transformation library for data structures. In *DSL*, pages 20–20, 1997.

28 Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *OOPSLA*, pages 561–578, 2014.

29 C. Stewart, Kai Shen, A. Iyengar, and Jian Yin. Entomomodel: Understanding and avoiding performance anomaly manifestations. In *MASCOTS*, pages 3–13, 2010.

30 The Grinder Java load testing framework. `http://grinder.sourceforge.net/`, 2013.

31 The SmartBear distributed testing framework. `http://support.smartbear.com/articles/testcomplete/distributed-testing-tutorial`, 2013.

32 M. Vandevoorde. *Exploiting Specifications to Improve Program Performance.* PhD thesis, Massachusetts Institute of Technology, 1994.

33 Jeannette M. Wing. Writing larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

34 Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *ISSTA*, pages 90–100, 2013.

35 Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *PLDI*, pages 419–430, 2009.

36 Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. LeakChaser: Helping programmers narrow down causes of memory leaks. In *PLDI*, pages 270–282, 2011.

37 Guoqing Xu, Nick Mitchel, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *PLDI*, pages 174–186, 2010.

38 Guoqing Xu and Atanas Rountev. Precise memory leak detection for Java software using container profiling. In *ICSE*, pages 151–160, 2008.

39 Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. In *PLDI*, pages 160–173, 2010.

40 Guoqing Xu, Dacong Yan, and Atanas Rountev. Static detection of loop-invariant data structures. In *ECOOP*, pages 738–763, 2012.

41 YourKit Java Profiling. `http://www.yourkit.com`, 2015.

42 Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces: A device-driver case. In *ASPLOS*, pages 193–206, 2014.

43 Pingyu Zhang and Sebastian Elbaum. Amplifying tests to validate exception handling code. In *ICSE*, pages 595–605, 2012.

44 Bowen Zhou, Jonathan Too, Milind Kulkarni, and Saurabh Bagchi. WuKong: automatically detecting and localizing bugs that manifest at large system scales. In *HPDC*, pages 131–142, 2013.

Hybrid DOM-Sensitive Change Impact Analysis for JavaScript

Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman

University of British Columbia
Vancouver, BC, Canada
`saba,amesbah,karthikp@ece.ubc.ca`

Abstract

JavaScript has grown to be among the most popular programming languages. However, performing change impact analysis on JavaScript applications is challenging due to features such as the seamless interplay with the DOM, event-driven and dynamic function calls, and asynchronous client/server communication. We first perform an empirical study of change propagation, the results of which show that the DOM-related and dynamic features of JavaScript need to be taken into consideration in the analysis since they affect change impact propagation. We propose a DOM-sensitive hybrid change impact analysis technique for JavaScript through a combination of static and dynamic analysis. The proposed approach incorporates a novel ranking algorithm for indicating the importance of each entity in the impact set. Our approach is implemented in a tool called Tochal. The results of our evaluation reveal that Tochal provides a more complete analysis compared to static or dynamic methods. Moreover, through an industrial controlled experiment, we find that Tochal helps developers by improving their task completion duration by 78% and accuracy by 223%.

1998 ACM Subject Classification D.2.5 Testing and Debugging, D.2.7 Distribution, Maintenance, and Enhancement

Keywords and phrases Change impact analysis, JavaScript, hybrid analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.321

1 Introduction

To remain useful, a software program must continually change to adapt to the changing environment [13]. Code change impact analysis (CIA) [3] aims at identifying parts of the program that are potentially affected by a change in the code. Impact analysis has been a popular research area [2, 5, 17, 19, 20]. Most of the research, however, is focused on traditional programming languages and ignores code, which is used extensively in modern web applications today.

JavaScript requires a novel approach for effective change impact analysis, because it has a set of unique features that makes it challenging to comprehend [1] and analyze by traditional code analysis techniques [8].

The first feature is the interplay between the JavaScript code and the Document Object Model (DOM) at runtime. The DOM is a standard object model representing HTML at runtime. DOM APIs are used in JavaScript for dynamically accessing, traversing, and updating the content, the structure, and the style of HTML pages. We have observed that the impact of a code change can be propagated through the DOM, even when there may be no visible connections between JavaScript functions and variables in the JavaScript code. The second feature pertains to the highly dynamic [21] and event-driven [25] nature of JavaScript code. For instance, a single fired event can dynamically propagate on the DOM tree [25] and

© Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 321–345

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

trigger multiple listeners indirectly. These implicit relations between triggered functions are not directly visible in the JavaScript code. Finally, XMLHttpRequest (XHR) objects used for asynchronous communication in JavaScript can transfer the impact of a change between two parts of the program that are not explicitly connected through the code. For instance, a server response can dynamically generate and execute JavaScript code on the client-side through callbacks.

Traditional impact analysis has been performed either by static analysis or dynamic analysis. However, the aforementioned challenges make it difficult for JavaScript static analysis techniques [8, 12, 22] to carry out impact analysis effectively. None of these techniques can provide support for the DOM-based, dynamic and asynchronous JavaScript features. Further, current dynamic and hybrid analysis techniques [26, 27] ignore the aforementioned challenges, i.e., they overlook the important role of the DOM in their analysis and they do not support the hidden relations that are created through event-handler registration, event propagation, and asynchronous client/server communication.

In this paper, we propose a hybrid analysis method for change impact analysis of JavaScript-based web applications that combines the advantages of both static and dynamic analysis techniques to obtain a more complete impact set. Our analysis is DOM-sensitive and aware of the event-driven, dynamic and asynchronous entities, and their relations in JavaScript. It creates a novel graph-based representation capturing these relations, which is used for detecting the impact set of a given code change. The main contributions of our work are as follows.

- A formalization of factors and challenges involved in change impact analysis for JavaScript.
- An exploratory study to investigate the existence and role of impact paths that require the analysis of DOM-related and event-based features in JavaScript. The results show that these features exist in real-world applications and cannot be ignored in proper change impact analysis for JavaScript.
- A DOM-sensitive event-aware hybrid change impact analysis technique for JavaScript. The approach creates a novel hybrid model that is used for identifying the impact set of a change in a given JavaScript application.
- A set of metrics for ranking the inferred impact set to facilitate the finding and understanding of the desired change impact by developers.
- An implementation of our approach in a tool called Tochal (TOol for CHange impact AnaLysis). Tochal is open source and available for download [24].
- An empirical evaluation of Tochal through a comparison with traditional pure static and dynamic analysis approaches, as well as a controlled experiment to assess the usefulness of Tochal in an industrial setting.

Our results show that event-driven and dynamic interactions between JavaScript code and the DOM are prominent in real applications, can affect change propagation, and thus should be part of a JavaScript impact analysis technique. We also find that a hybrid of both static and analysis techniques is necessary for a more complete analysis. And finally, Tochal can improve the performance of developers in terms of both impact analysis task completion duration (by 78%) and accuracy (by 223%).

2 Impact Transfer in JavaScript

Many unique features of JavaScript applications require special attention during impact analysis. These features include (but are not limited to) DOM interactions, dynamic event-driven execution of functions, and asynchronous communication with the server. The impact

```
1  function checkPrice() {
2    var itemName = extractName($('#item231'));
3    var cadPrice = $('#price_ca').innerText;
4    $.ajax({
5       url : "prices/latest.php",
6       type : "POST",
7       data : itemName,
8       success : eval(getAction() + "Item")
9    });
10   confirmPrice();
11 }
12 function updateItem(xhr) {
13   var updatedInfo = getUpdatedPrice(xhr.responseText);
14   suggestItem.apply(this, updatedInfo);
15 }
16 function suggestItem() {
17   if (arguments.length > 2) {
18     displaySuggestion(arguments1);
19   }
20 }
21 function calculateTax() {
22   $(".price").each(function(index) {
23     $(this).text(addTaxToPrice($(this).text()));
24   });
25 }
26 $("#price-ca").bind("click", checkPrice);
27 $("prices").bind("click", calculateTax);
```

Figure 1 Motivating example: JavaScript code.

```
1  <img id='item231' src='img/items/231.png'
                itemName='dress'/>
2  <fieldset name='prices'>
3       <div class='price' id='price-ca'>120</div>
4       <div class='price' id='price-us'>110</div>
5  </fieldset>
```

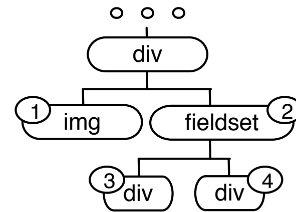


Figure 2 Motivating example: HTML/DOM.

can be transferred through these entities, without direct visible relations in JavaScript. Throughout the rest of the paper, we use the term *indirect* impact to refer to change impact transferred through such features. Impact transferred directly through JavaScript code, e.g., through function calls, is referred to as *direct* impact.

▶ **Definition 1** (Relevant Entities). Let f be a JavaScript function, d a DOM element, and x an XHR object. If F, D, and X are sets representing each of those entities, respectively, then the set of all relevant entities is defined as $E : \sum \varepsilon \leftarrow F \cup D \cup X$

Change impact can propagate between these JavaScript entities through a series of *read* and *write* operations.

▶ **Definition 2** (Read/Write Operations). Let ε_1 and ε_2 be arbitrary entities in E. Suppose ε_1 writes to ε_2 at time τ. Then the relation is represented as $\varepsilon_1 W_\tau \varepsilon_2$. If ε_1 is read by ε_2 at time τ, then the relation is represented as $\varepsilon_1 R_\tau \varepsilon_2$.

The semantics of the relations between entities are drawn from actual JavaScript execution mechanisms (e.g., function f reads from a DOM element d). However, for each W/R relation between two entities, there is a conceptual R/W relation between the same two entities in

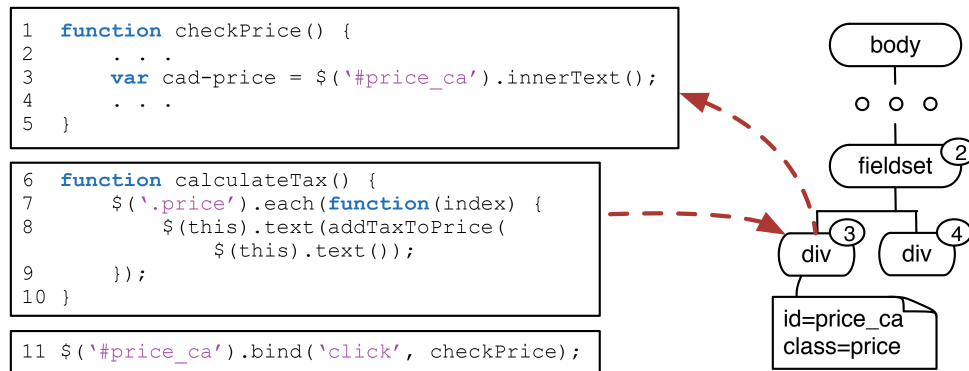


Figure 3 Impact transfer through DOM elements.

the opposite direction (e.g., d writes to f). Definition 3 formalizes the notion of impact transmission between two entities.

▶ **Definition 3** (Impact). Let ε_1 *and* $\varepsilon_2 \in E$. If the value and/or the behaviour of ε_2 depends on the value and/or the behaviour of ε_1, then ε_1 is said to have an impact on ε_2, represented as $\varepsilon_1 \rightarrow \varepsilon_2$.

The impact can also be indirectly transferred from entity ε_1 to ε_2, if ε_1 writes to ε_3, which is later read by ε_2. We call such this relation a `WR` pair.

We use a simple motivating example, presented in Figures 1–2, to illustrate the challenges and explain each definition. We use different portions of this example in the following subsections. Note that this is a simple example and these challenges are much more potent in large and complex web applications.

2.1 Impact through the DOM

In modern web applications, the DOM structure [7] evolves dynamically through the execution of JavaScript code, to update the content, structure, and style of the application in a responsive manner. A JavaScript function can write to a DOM element, which in turn can be read by another function and thus impact its behaviour. Such DOM elements can transfer the impact of a change indirectly. Impact transferred through the DOM introduces an important challenge in identifying change impact in web applications. Hence, in this work, we propose DOM-related dependencies as additional means of impact transfer.

▶ **Example 4.** Figure 3 displays a portion of the motivating example that contains a hidden DOM-based dependency. Function `calculateTax()` retrieves all DOM elements having the class attribute `price` (line 7). The function then recalculates the price of each element to include the tax and rewrites the value of the element with the new price (line 8). Later, when the function `checkPrice()` (line 1) is invoked through a user event (registered in line 11), it retrieves the value of a DOM element with id `"price-ca"` (line 3) and uses this value to perform other operations. So far, there is no direct relation between functions `calculateTax()` and `checkPrice()` that shows any dependency between the two code segments. However, looking at the DOM structure shown on the right side of Figure 3, we can see that the element with ID `"price-ca"` is also an instance of the `price` class (element ③ on the DOM tree). This means that the value used by `checkPrice()` may be affected by `calculateTax()`. This is a simple example of dynamic DOM dependency, which needs to be taken into account in impact analysis of JavaScript applications.

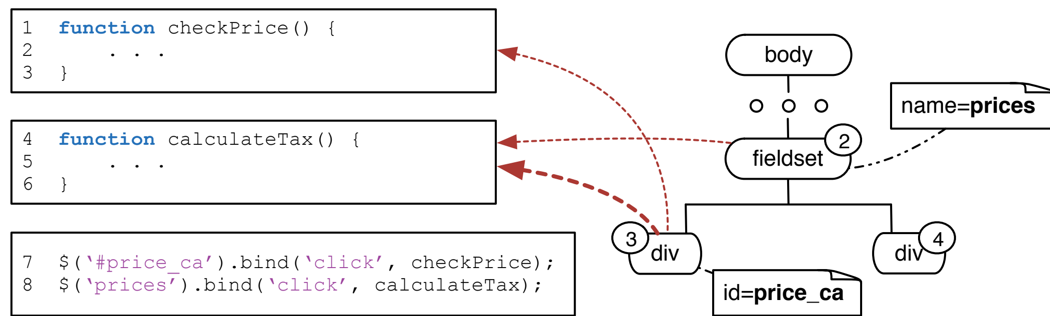


Figure 4 Impact transfer through event propagation.

2.2 Impact through Event Propagation

In web applications, a single event can propagate on the DOM tree and invoke multiple handlers of the same event-type attached to any of the ancestors of the target element [25]. The direction of the event propagation depends on whether the capturing or bubbling mode is enabled. When *capturing* is enabled, the event is first captured by the parent element and then passed to the event handlers of children, with the deepest child element being the last. With *bubbling* enabled, an event first triggers the handler of the target element on which the event was fired, and then it bubbles up and triggers the parents' handlers. The second type of impact dependencies we introduce pertain to the hidden relations between the handlers invoked via propagation of the original event on the target DOM tree. Such invoked handlers can be involved in change impact propagation, i.e., they can affect the control flow of the application and thus need to be considered in impact analysis for JavaScript.

▶ **Example 5.** In a segment of the motivating example shown in Figure 4, `checkPrice()` is attached to the element with id `price-ca` as an event handler (line 7). Therefore, if a user clicks on that element (element ③ on the right side DOM tree), function `checkPrice()` gets invoked. However, `price-ca` is contained within a `fieldset` element with the name `prices` (element ② on the DOM tree), which is similarly bound to an event handler for the `click` event (Figure 4, line 8). Due to event *bubbling*, any click on `price-ca` will bubble up to `prices` and trigger its event handler as well. Hence, `calculateTax()` is invoked through propagation of an event originally targeting `price-ca`. As a result, the execution of `calculateTax()` also depends on the `price-ca` element, in addition to the `prices` element.

2.3 Impact through Asynchronous Callbacks

XMLHttpRequest (XHR) objects help developers enrich user experiences with web applications through asynchronous communication with the server. While increasing the interactivity and responsiveness of applications, XHR usage adds additional complexity to impact analysis. Each XHR consists of three main phases: *open*, *send*, and *response*. A callback function is invoked when the XHR response is received from the server, without a user involvement. A change in opening the request, sending it, or the sent message can impact the response of the server, as well as the behaviour of the application after receiving the response through a callback function. Different components of XHR objects can make the detection of control and data flow relations more troublesome, particularly when these components are not necessarily collocated in the same function or module. This motivates the third type of impact dependencies we introduce in this work.

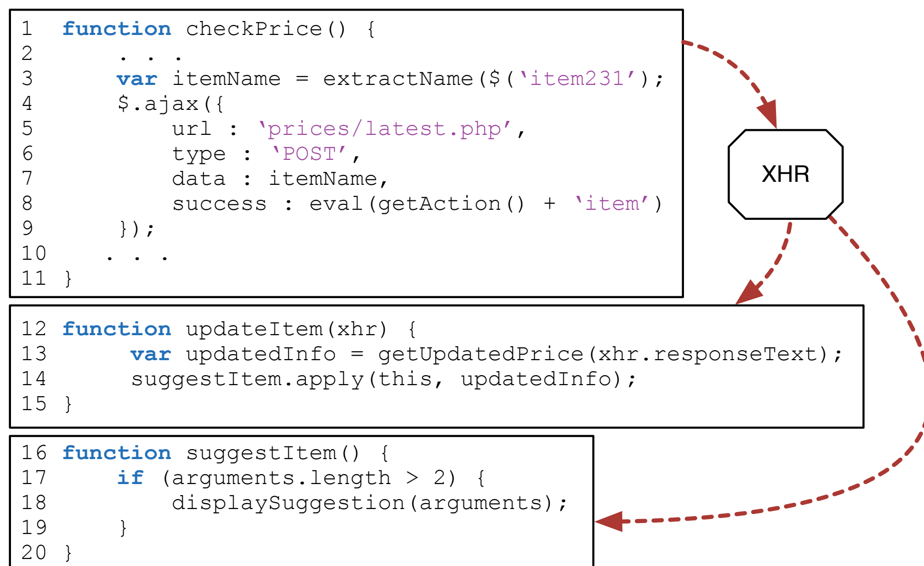


Figure 5 Impact transfer through asynchronous callbacks.

▶ **Example 6.** `checkPrice()` sends an asynchronous request to the server (lines 4–9 in Figure 5). However, the assigned callback function cannot be recognized statically, as the code uses the action chosen by the user dynamically to invoke the appropriate function (line 8, `eval`). Let's assume that in this example the selected action is to "update" the price of an item, and hence the `updateItem()` function is assigned as the callback function of the XHR object. As it can be seen, there are no direct function calls or shared variables between `checkPrice()` and `updateItem()` to enable traditional change impact analysis techniques to derive a dependency relation between the two functions. However, the XHR message along with the data that was sent with it can affect the response that comes back from the server and thus can impact the behaviour of `updateItem()`.

2.4 JavaScript Dynamism

Many traditional impact analysis techniques use static aspects of the code to determine the impact set of a change. Dynamic features of the JavaScript language pose a challenge to static analysis techniques. For instance, almost everything in JavaScript, from fields and methods of objects to their parents, can be created or modified at runtime. Also, JavaScript's dynamic policies for invoking functions can add more complexities. One such policy is function variadicity, which is common in web applications [21]; i.e., in JavaScript, functions can be invoked with more or less arguments compared to the parameters specified in a function's static declaration. In addition to the DOM, event, and XHR challenges, the dynamic features of the JavaScript language need to be addressed in an effective change impact analysis technique.

▶ **Example 7.** In line 14 of Figure 5, `updateItem()` invokes `suggestItem()` through the `apply()` function, which makes it impossible to infer the number of passed arguments statically. Function `suggestItem()`, the callee, takes no arguments according to its declaration (line 16). Yet, the function is invoked with an arbitrary non-zero number of arguments, which can change the execution of the application (Figure 5, line 17). Knowledge of the passed arguments at runtime is crucial for performing precise data-flow analysis, required for impact analysis.

2.5 Impact Paths

The concept of `WR` pairs can be generalized to any mechanism that can transfer impact in JavaScript, such as XHR objects, function arguments, and function return values. Consecutive `WR` pairs involving all JavaScript entities can form general impact paths, as described in Definition 8.

▶ **Definition 8** (Impact Path). An impact path of an entity ($P(\epsilon)$) is a directed acyclic path starting from entity ϵ. The nodes on the path are entities in the system, and the edges are the directed impact relations that connect those entities.

For instance, `updateItem()` → `suggestItem()`, `checkPrice()` → `#price-ca` (DOM element with id=`price-ca`), `checkPrice()` → `#price-ca` → `calculateTax()` → `addTaxToPrice()` are examples of impact paths that exist in the running example (Figure 1).

The length of an impact path is defined as the number of entities in the path. The minimum length for propagation of the impact of a change through DOM elements is 3 ($f\ W_{\tau_1}\ d\ R_{\tau_2}\ g \mid \tau_1 < \tau_2,\ d \in D,\ f, g \in F$).

3 Exploratory Study: DOM-related and Event-based Impacts

We conducted an exploratory study to investigate the role of JavaScript's DOM-related, event-based and dynamic features in code change propagation. Our goal was to understand whether DOM elements, event handlers, and propagated events contribute to forming new impact paths in JavaScript code.

Subject Applications. We selected ten web applications that make extensive use of JavaScript on the client-side for this study. We selected these applications from (1) participants of recent JavaScript programming contests, and (2) trending and popular JavaScript projects on GitHub[1]. They are listed in column 1 of Table 1.

Design. We capture JavaScript-DOM interactions as well as any occurrences of event propagation on the DOM tree. For each DOM access that occurs during the execution, we collect the accessed entity, the JavaScript function that accesses the DOM, and the access type. Access types are directed operations performed on DOM elements by JavaScript functions, where the direction of the access is determined by the direction of the flow of data. For instance, assume function foo creates DOM element e at time τ, which means $foo\ W_\tau\ e$. Then the type of access is `element-creation` and the direction of access is from foo to e. The collected data is analyzed to extract the impact set for each of the JavaScript functions involved in the execution. The DOM elements that are located on at least one impact path of a function are the ones that can contribute to the impact propagation process and are called `WR` elements for simplicity. The considered impact paths are required to have a length of at least three (e.g., $fooW_{\tau_1}eR_{\tau_2}bar$, for functions foo and bar and DOM element e). Moreover, redundant impact pairs (reads and writes between same entities) do not contribute to the impact paths and are therefore eliminated from the analysis.

[1] `https://github.com/trending/`

Table 1 (A) Results of analyzing JavaScript's DOM-related and dynamic features. (B) Factors in determining impact metrics.

JavaScript		(A) DOM and dynamic features				(B) Factors of impact metrics				
Application	LOC	# DOM elem.	% WR DOM elem.	# of handlers	% prop. handlers	Avg. Path Length	fan-in elem.	fan-out elem.	fan-in func.	fan-out func.
same-game	229	62	98	20	45	6.3	1.9	2.9	23.1	15.2
ghostBusters	343	44	61	39	0	4.3	3.3	0.4	2.6	20.0
simple-cart	9238	41	51	14	0	3.9	2.1	1.7	2.7	3.3
mojule	522	47	17	18	33	7.0	1.5	2.1	4.6	3.3
jq-notebook	839	1	100	21	38	4.0	16.0	11.0	0.9	1.3
doctored.js	3534	2	50	47	15	5.3	4.0	7.0	1.0	0.8
jointlondon	2498	34	9	16	0	3.7	0.8	2.2	1.6	0.6
space-mahjon	983	61	10	53	4	4.0	1.8	3.0	1.5	0.9
listo	354	5	20	10	0	4.0	0.1	1.5	21.4	1.9
peggame	1274	17	6	23	0	3.0	0.8	1.3	4.3	2.1
Average	1981	31	**42**	26	**14**	**4.6**	**3.2**	3.3	**6.3**	**4.9**

Results. Each application is manually exercised in different scenarios multiple times and the results are integrated. The results are shown in section (A) of Table 1. The first column of section (A) displays the total number of DOM elements accessed by JavaScript code during execution. The second column of the section shows the ratio of WR DOM elements to the total number of involved DOM elements from the first column. The number of DOM event handlers that were triggered during the execution is shown in column three. Column four represents the percentage of handlers that were invoked through event propagation (capturing or bubbling) to the total number of triggered handlers. The average length of impact paths in each application is depicted in column one of section (B).

The results show that on average, 42% of the DOM elements that were accessed during the execution of these applications, were part of an impact path between two functions. Moreover, about 14% of the executed event handlers were invoked through event propagation mechanisms. The results thus reveal the importance of DOM elements in transferring the impact. Also, the role of propagated event handlers is significant in determining the dynamic behaviour of a JavaScript application. Hence, a CIA technique for JavaScript application should consider the DOM-related and event-based features as media for propagating the impact.

We further analyze the structure of the created dependency graphs to gain more insight into the nature of DOM- and event-based relations within JavaScript applications. Among all structural and semantic aspects of the graphs, the average fan-in and fan-out scores of the functions and DOM elements in the graphs are reported in section (B) of Table 1. These factors are selected due to their correlations with the ratio of WR elements in subject applications. We use this information later in the paper, when we propose a set of metrics for ranking the impact set (Section 5).

4 Hybrid Analysis

We propose a hybrid technique, called Tochal, which augments static analysis with dynamic analysis to enable a DOM-sensitive and event-aware change impact analysis method for JavaScript applications.

4.1 Static Control-Flow and Partial Data-Flow Analysis

Our approach first identifies JavaScript entities that *can* be analyzed statically. Among entities described in Definition 1, JavaScript functions (F) are the only entities that fit

this criterion. The DOM is created and mutated during execution. This limits the static reasoning about its structure and possible event propagations that would affect change impact. Regarding XHR objects, it is not easy to infer statically what messages are received from the server. Moreover, there is no static information available regarding the order and timing of asynchronous callbacks.

Our static analysis module incorporates *direct* relations between functions into a static call graph (SCG) by analyzing the JavaScript code. In JavaScript, functions are first-class citizens and receive the same treatment as objects; we augment the same static call graph with global variables, which we treat similarly as the functions.

To increase the precision of the static analysis, which in turn improves the quality of the impact set, we perform a pruning algorithm on the extracted dependencies. The pruning is conducted based on a partial data-flow analysis of the call graph. Function invocations are not considered as impact relations unless the two functions have a data dependency through passed arguments or return values, as described in Definition 9. This does not concern data dependencies through global variables shared between two functions, where separate dependency relations are formed.

▶ **Definition 9** (Function Dependencies). Let $\rho, \delta \in P$. Then impact relations between f and g are defined as:

- $f \rightarrow g$ if f invokes g and the signature of g indicates that it takes parameters.
- $g \rightarrow f$ if f invokes g and the definition of g includes a return value.

4.2 Analyzing the Dynamic Features

To include the dynamic features of the JavaScript language in our impact analysis, our dynamic analysis module intercepts, transforms, and instruments the JavaScript code on-the-fly to collect execution traces. To collect a trace of function executions, the beginning and the end of each JavaScript function are instrumented. Function declarations are modified to collect traces of function invocation and passed arguments. We also trace function terminations and return statements (if they exist in the function). These traces are then used to create a dynamic call graph (DCG) that captures dependencies between function executions at runtime.

The DCG is an under-approximation of the call graph, while the SCG is an over-approximation of the call graph. The DCG contains fewer false positives compared to the SCG, and we augment it to capture DOM-related, event-driven, and asynchronous features of JavaScript, as explained below.

DOM-Sensitive Impact Analysis

A JavaScript function can impact a DOM element and vice versa, which is defined as:

▶ **Definition 10** (Direct Impact between JavaScript and DOM). Consider a DOM element d ∈ D, and a function f ∈ F. Then f can directly impact d and vice versa:

$$\begin{cases} f \rightarrow_\tau d & \text{if } f W_\tau d \\ d \rightarrow_\tau f & \text{if } f R_\tau d \end{cases}$$

Furthermore, the impact can travel from function f to function g, through a DOM element d, under certain conditions as defined in the next definition.

▶ **Definition 11** (Indirect JavaScript Impact through DOM)**.** Consider two functions f, g ∈ F, and a DOM element d ∈ D. f can indirectly impact g through d, if and only if the following conditions hold:

$$f \rightarrow_d g \text{ if } \begin{cases} fW_{\tau_1}d & \& \\ gR_{\tau_2}d & \& \\ \tau_2 > \tau_1 \end{cases}$$

In other words, function f can potentially impact function g through DOM element d, if f writes to d and g reads from the same element. Such a write-read (`WR`) pair indicates the existence of a potential impact between the two functions, if the read instruction happens after the write. Such `WR` pairs can occur subsequently, involving more elements and functions in the application. The reading function can itself write to a DOM element and augment the propagation path. The same change can then potentially impact all elements and functions that are on such a path.

To analyze how the DOM transfers the change impact (Definition 11), all *read* and *write* accesses to the DOM need to be monitored dynamically. Each access is made from a JavaScript function to a DOM node, element, or an attribute, and through standard DOM API calls (e.g., `getElementById`, `querySelector`). We modify the prototype of the `Node`, `Document` and `Element` classes to be able to dynamically intercept DOM accesses, while preserving the original behaviour of these classes. This allows us to monitor changes to the structure of the DOM tree, as well as the existence, content, and attributes of DOM elements.

It is worth mentioning that the caller functions are extracted from the dynamic context of the intercepted DOM API calls. As a result, if a function does not exist or is not detected statically (e.g., it is created through an `eval` statement), it will still be captured in the dynamic phase if it interacts with other entities and is part of the execution.

For each function and DOM element involved in an access, we augment the dynamic call graph by adding two nodes (if they do not already exist), and connecting them through an edge representing the type of access that was made from the function to the element.

Event-Based Impact Propagation

TOCHAL also captures all event handlers called directly and indirectly through event propagation on the DOM tree.

Definition 12 summarizes the potential impact transmission between a DOM element and all event handlers that are called through event propagation.

▶ **Definition 12** (Indirect Impact via Propagation)**.** Let d be a DOM element that has an event handler for event e. Consider $prop_e[d]$ to be the set of all JavaScript functions that are submitted as handlers for event e to d or any of its ancestors in the DOM tree, and thus can be triggered by event propagation. Then d can impact all these handlers indirectly: $d \rightarrow prop_e[d]$.

Moreover, the dynamic analysis module yields information on function arguments and return values for all directly and indirectly invoked functions. These variables may differ from what has been declared in static function signatures, due to function variadicity in JavaScript.

Table 2 Impact transfer through different entities.

Assume $f, g \in F$ (functions), $d \in D$ (DOM) and $x \in X$ (XHR)	
Relation	**Description**
$fW_{\tau}g$	1. f calls g and passes arguments. 2. g calls f and f returns a value.
$fW_{\tau}d$	1. f creates element d, adds it to the DOM tree, deletes it, or detaches or relocates it from the DOM. 2. f modifies the content or the attributes of d.
$dR_{\tau}f$	1. f uses information regarding the content, attributes, or location of d in the DOM. 2. f is bound to d through an event handling mechanism. 3. f is set as an event handler of one of the ancestors of d, that can be triggered via event propagation.
$fW_{\tau}x$	1. f opens x as a new XHR object. 2. f sends a previously-created XHR object x.
$xR_{\tau}f$	1. f is set as the callback function of x 2. f sends a previously-created XHR object x.

XHR Relations

There are three main phases in the lifecycle of each XHR object: open, send, and response. These three phases can be scattered throughout the code. In addition, callbacks from the server-side could invoke other functions on the client-side, and hence it is not trivial to find the XHR components statically. Our technique instruments and intercepts each component of an XHR object by wrapping around the XHR object of the browser. The gathered information is then used to augment the dynamic call graph. Similar to the previous features, new nodes representing XHR objects are added to the graph, the involved functions nodes are only added if they were not previously included, and the function nodes are linked to the XHR node based on the type of access they make to the XHR object. The access types are defined by the type of the interaction between a function f and an XHR object x, and determine the direction of the impact relation. For instance, if f creates and opens x, then $f \rightarrow x$, while if f is registered as the callback function of x, then $x \rightarrow f$.

Similar to the static call graph, we enhance the dynamic call graph using inter-procedural data-flow analysis. Arguments and return values of functions are used to trim the call graph where there is no data flow between two functions (Definition 9). However, instead of using the static function code, the dynamic arguments and return values are used to support function variadicity at run time.

4.3 Hybrid Model for Impact Analysis

At this stage, TOCHAL creates a system dependency graph by integrating the obtained static and dynamic call graphs. This graph is used for performing impact analysis on JavaScript applications. The dynamic part of the model contributes to the precision of the analysis, while its static features make it more complete. We take a best-effort approach for fulfilling soundness, following the soundness manifesto [14]. In order to satisfy the practicality of our approach in terms of precision and scalability, complete soundness is not a concern of our approach. The hybrid model is represented as a directed graph. The vertices are system entities and the edges are the potential impact relations as summarized in Table 2.

Vertices. The vertices in the graph are all entities that are present statically or are created during the execution of an application. The vertices can take one of the following types:

- *JavaScript functions.* JavaScript functions extracted by our static analyzer (Section 4.1)

are added as vertices. Functions found dynamically are added as well due to their involvement in the impact propagation, even if they are not connected directly (Section 4.2).

- *DOM elements.* The importance of DOM elements in transferring the impact dynamically was discussed in Section 4.2. Accessed DOM elements (and their contextual information) are captured as vertices in the model.
- *XHR objects.* XHR vertices incorporate information regarding the creation and sending of messages, as well as callback functions and data transmitted dynamically between the server and the browser.

Edges. The edges in the graph are labeled and directed.

- *Direction.* The direction of each edge depicts the flow of the data between two vertices. The edges are categorized into *read* and *write* accesses. An edge is directed from the vertex that writes (offers) the data to the vertex that reads it.
- *Labels.* The edge labels indicate the type of dependency relations that connect the vertices. Different labels are used to connect different vertices, since the valid operations vary for each category of vertices.

▶ **Example 13.** Figure 6 shows a dependency graph for the running example (Figures 1 and 2) obtained through the static analysis module alone. On the other hand, Figure 7 depicts a simplified hybrid graph utilizing both the static and dynamic analysis modules of TOCHAL. This is hard to do with the static graph. However, using the hybrid graph, one can find the potential impact set of a change in entity ε by tracing the graph forward, starting from ε.

Consider a case where a developer plans to make a change to the `calculateTax()` function (line 21 of Figure 1), and would like to find the potential impact before making the actual change. A DOM-agnostic static change impact analysis method (see Figure 6) would report that only `addTaxToPrice()` would be affected. The source code shows that next to the `addTaxToPrice()` function, DOM elements with class=`price` (lines 22–23 of Figure 1) can also be affected.

However, our hybrid DOM-sensitive analysis reveals that there exist more impact paths. The DOM element with id=`price-ca` is also a member of a class=`price` (box 4 of Figure 7). This element can thus be impacted by `calculateTax()`, and in turn can propagate the impact to `checkPrice()` indirectly (lines 3 & 26 of Figure 1, box 1 of Figure 7). Furthermore, evaluating the response of an XHR object, `checkPrice()` can then transfer the impact to `updateItem()` (box 6), which can propagate the impact to more functions (boxes 5, 8 & 9). To summarize, as our hybrid model shows, changing the `calculateTax()` function can affect six more elements in addition to the two elements that can be detected by statically analyzing the code. Thus, the proper impact set consists of functions `addTaxToPrice()`, `checkPrice()`, `updateItem()`, `getUpdatedPrice()`, `suggestItem()`, `displaySuggestion()`, the DOM element with id=`price-ca`, and the anonymous XHR object.

5 Impact Metrics and Impact Set Ranking

An impact set inclusive of the contributions of both static and dynamic analyses can become large and overwhelm the user. Considering that not all entities in the impact set are equally important, providing a ranking mechanism is essential for helping developers identify relevant impacted entities more efficiently.

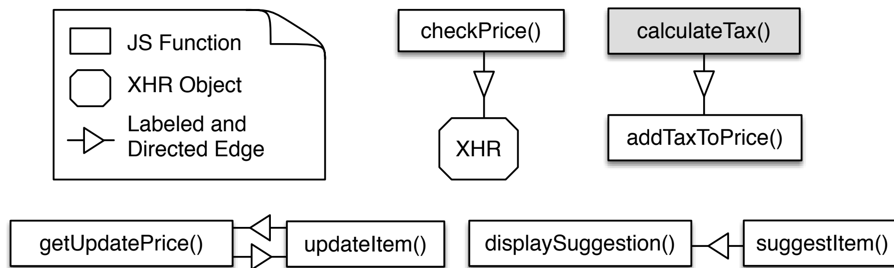


Figure 6 A static call graph, displaying the dependencies extracted from the running example (Figures 1 and 2).

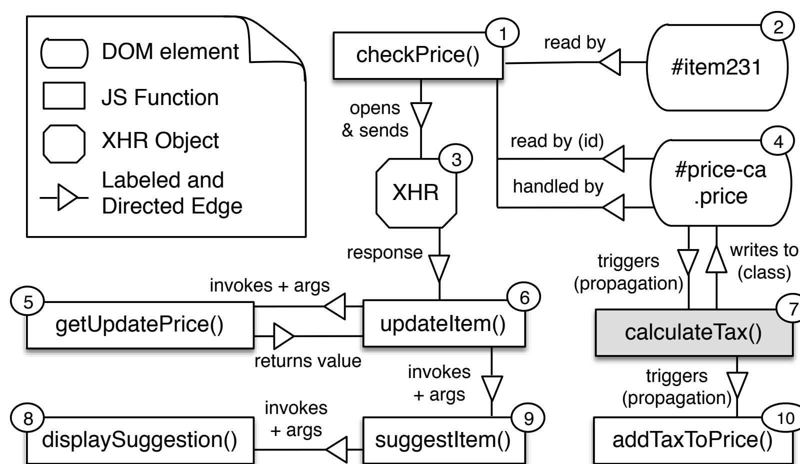


Figure 7 A sample hybrid graph, including the dynamic and DOM-sensitive dependencies extracted from the running example (Figures 1 and 2).

We propose a set of *impact metrics* to estimate the importance of each entity in the produced impact set. The impact metrics, outlined in Table 3, are variables derived from the semantic and structural characteristics of the hybrid graph. These metrics can affect the probability of impact propagation, through DOM-related and dynamic mechanisms of JavaScript. Based on the impact metrics, we propose an impact ranking mechanism as outlined in Definition 14. The *impact rank* score of each entity ϵ, referred to as $IR(\epsilon)$, is an estimation of the importance of ϵ in propagating the change, relative to other elements in the impact set.

▶ **Definition 14** (Impact rank). Let ϵ be an entity in the impact set. Then the impact rank of ϵ is defined as:

$$IR(\epsilon) = \frac{IR_{pr}(\epsilon) * \widehat{L}[P(\varepsilon)] * Fan_w(\epsilon)}{D_m(\epsilon)}$$

where, the value of the impact rank of an element in the impact set depends on four variables. (1) $IR_{pr}(\epsilon)$: the accumulation of impact ranks of immediate precedents of entity ϵ in the hybrid graph that are on an impact path from the change set to ϵ. If the impact is transferred

Table 3 Impact Metrics.

Entity	Metric	Description	CC with % WR DOM elem.
$d \in D$	$f_{in}(d)$	Number of functions f such that $fW_\tau d$	0.66
$f \in F$	$f_{in}(f)$	Number of elements d such that $fR_\tau d$	0.74
	$f_{out}(f)$	Number of elements d such that $fW_\tau d$	0.62
$\epsilon \in DorF$	$\widehat{L}[P]$	Average length of impact paths in the application	0.59
	$D_m(\epsilon)$	Minimum distance of ϵ from the change set	–

to ϵ from entities with higher ranks, then the importance of ϵ in transferring the impact potentially increases. (2) $D_m(\epsilon)$: the shortest distance of ϵ from the change set in the hybrid graph. The closer ϵ is to the change set, the higher is the probability of being impacted by the change in practice. Hence, a longer distance of ϵ from the change set has a lesser effect on its impact rank. (3) $\widehat{L}[P(\varepsilon)]$: the average length of an impact path starting from entity ϵ. If ϵ can cascade the impact to more elements and deeper levels in the propagation graph, then ϵ is potentially an important entity in the impact set. (4) $Fan_w(\epsilon)$: the weighted fan-in / fan-out score of functions and DOM elements in the hybrid graph is indicative of the number of entities that can impact and be impacted by ϵ directly; hence, we consider it as a determining factor in impact rank determination. $Fan_w(\epsilon)$ is calculated as follows:

$$Fan_w(\epsilon) = \begin{cases} w_1 * f_{in}(\epsilon) & \text{if } \epsilon \in D \\ w_1 * f_{in}(\epsilon) + w_2 * f_{out}(\epsilon) & \text{if } \epsilon \in F \end{cases}$$

$\widehat{L}[P(\varepsilon)]$ and $Fan_w(\epsilon)$ are extracted from the findings of our exploratory study (Section 3). During the course of the study, we measured the semantic and topological properties of hybrid graphs of ten web applications. Among the analyzed variables, the length of impact paths, fan-in of DOM element node, and fan-in and fan-out of function nodes (section (B) of Table 1) have a correlation with the number or ratio of DOM elements that can transfer the impact. Hence, these variables can affect the probability of impact propagation through an entity. They thus play a role in these two determining factors influencing the overall impact rank.

6 Tool Implementation: Tochal

We implemented TOCHAL using JavaScript libraries such as Esprima,[2] Estraverse[3] and Escodegen[4] for parsing and transforming the JavaScript code. We use these libraries to instrument functions in a manner that permits us to collect data about function invocations, function entries and function exits. The collected data also include dynamic values of functions' arguments and return values. External files are attached to the beginning of each document that allow instrumentation and interception of DOM events, XHR objects and timeouts. We use the Mutation Summary library [16] to detect JavaScript code appended to the DOM on the fly. Therefore, we can extend function instrumentation to the JavaScript code that gets created dynamically during application execution. We use WALA [23] to extract the static call graphs of applications.

TOCHAL provides an interface for developers to utilize our hybrid change impact analysis. The main goal is to facilitate the comprehension of change impact "during" development and

[2] http://esprima.org
[3] https://github.com/Constellation/estraverse/
[4] https://github.com/Constellation/escodegen/

debugging activities. Hence, the analysis needs to be available to developers while they make changes to their application. We have integrated our impact analysis method within Google Chrome's DevTools [10], a popular web development environment. This decision entails a number of benefits, namely (1) the approach is complementary to existing web development platforms and environments; it does not change the functionality they provide, but augments their capabilities. (2) The developer can perform the impact analysis in the same context as the code, and can preserve her mental model of the code. (3) The developer is not required to learn a new tool, or divide her attention between two different tools.

The interface allows the user to select JavaScript functions (including XHR callbacks) or DOM elements as the change set, and then perform the impact analysis. Chrome's DevTools includes a set of panels, each providing a window to a subset of functionalities that the platform provides. Two panels are of more interest for us: "elements" and "sources". The elements panel visualizes and provides inspection mechanisms on the DOM. The sources panel displays all of the JavaScript code that contributes to the application. We add a sidebar to each of these panels, allowing the user to invoke the CIA unit, on a selected entity, at any stage of the development. Tochal is publicly available for download [24].

7 Evaluation

We empirically evaluate the effectiveness and usefulness of Tochal through the following research questions:

RQ1 How does our hybrid method compare to static/dynamic analysis methods?
RQ2 Does Tochal help developers in performing change impact analysis in practice?

We address these questions through two studies, each described in the following two subsections, respectively.

7.1 Study 1: Comparing Static, Dynamic, and Hybrid Analyses

To address RQ1, we conduct a study to evaluate the impact sets extracted using Tochal in comparison with those detected by static and dynamic analysis techniques separately. We compare Tochal with a state-of-the-art static analysis technique. We also examine the differences in the outcomes of Tochal with those of the dynamic analysis unit of Tochal. The term *dynamic* analysis encompasses the DOM-sensitive, dynamic, event-driven, and asynchronous analysis performed by Tochal. Our first hypothesis is that Tochal outperforms static impact analysis due to its support for *dynamic* analysis. We also hypothesize that while dynamic analysis is a significant part of Tochal, it is outperformed by tochal's hybrid analysis.

We decided to compare Tochal with static and dynamic approaches, since to the best of our knowledge, Tochal is the first change impact analysis technique for JavaScript and there is no similar tool available for JavaScript.

Design and Procedure

The only entities that can be analyzed by both static and *dynamic* analysis methods are JavaScript functions. Hence, to be fair to both static and dynamic analyses, we configure Tochal to only deliver functions in the impact sets. Tochal's hybrid model and analysis, however, do not differ from what is described in Section 4 and use DOM-based, dynamic and asynchronous entities and relations to extract the functions in the impact sets.

Table 4 Results of comparison between static, dynamic and TOCHAL (RQ1) (A) Comparison of impact sets (B) Comparison of functions in system dependency graphs.

Application	(A) Impact sets											(B) Functions				
	Tochal			Static			Static/Tochal	Dynamic			Dynamic/Tochal	Tochal	Static/Tochal	Dynamic/Tochal	Pure Static	Pure Dynamic
	avg	min	max	avg	min	max	%	avg	min	max	%	avg	%	%	%	%
same-game	4.33	2	7	0.67	0	1	15	3.67	2	6	85	16	56	93	6	44
ghostBusters	1.33	0	2	0.33	0	1	25	1.33	0	2	75	10	80	100	0	20
simple-cart	1.67	0	3	0.67	0	1	40	1.67	0	3	100	44	74	91	9	26
mojule	1.00	0	2	0.33	0	1	33	0.67	0	2	67	21	24	90	10	71
jq-notebook	2.67	0	7	0.67	0	2	25	2.33	0	6	87	19	47	100	0	53
doctored.js	1.67	0	4	0.33	0	1	20	1.33	0	3	80	38	67	50	56	33
jointlondon	2.33	0	5	0.67	0	1	29	1.67	0	4	72	36	31	85	15	69
space-mahjon	2.67	1	5	1.00	0	2	37	2.00	0	5	63	27	56	93	7	44
listo	1.33	1	2	0.00	0	0	0	1.33	1	2	100	12	75	58	25	42
peggame	2.67	1	6	1.00	1	1	37	2.00	0	5	75	24	83	75	25	17
Average	**2.07**	0.5	4.3	**0.57**	0.1	1.1	**26**	**1.8**	0.3	3.9	**80**	**25**	**59**	**84**	**15**	**42**

We use the same set of subject applications from our exploratory study of Section 3 (Table 4, column 1). For each application, we randomly sample three functions and extract their impact sets using each of the methods. We compare the impact sets to assess the completeness of the outcomes of analysis with each approach. The sample functions are selected from a pool of functions that are recognized by all three analysis methods. In other words, static and *dynamic* analysis alone are not able to detect some functions that are detected by TOCHAL and are involved in its hybrid analysis. If static/*dynamic* analysis is performed on any of the functions it does not recognize, the impact set will be empty. We aim at comparing impact sets at this stage and these functions are unable to provide useful information regarding the analysis. Thus, we do not include such functions in the comparison. However, this indicates the need for an investigation of the functions involved in each type of analysis (in the dependency graphs), as described in the next paragraph.

We measure the number of functions that are included in the dependency graphs of each analysis, contributing to the detection of the impact sets. The average number of functions in each type of analysis denotes the extent of the analysis performed for extracting the impact sets. Moreover, the lower number of recognized functions by an analysis method means that there are more functions for which the method is unable to perform CIA.

Pure Static Analysis. We use the static analysis part of our approach, which is built using WALA [23]. WALA is a leading static analysis tool for JavaScript, used by many other JavaScript analysis techniques [26, 27, 22]. It should be noted that WALA by itself does not support change impact analysis. For the purpose of this evaluation, we extended and directed it towards performing static impact analysis to conduct the comparisons.

Pure Dynamic Analysis. We disable the static module of TOCHAL and only utilize the *dynamic* analysis module. The applications are exercised through their test suites when available and manually within multiple sessions and the results are integrated. Each manual session follows a set of pre-defined scenarios that covers all main use-cases of the application that are accessible to an end-user.

Hybrid Analysis. We use the hybrid model of TOCHAL for performing impact analysis and obtaining the set of functions that are involved in the hybrid analysis.

Results and Discussion

To answer RQ1, we discuss the outcomes of the study, summarized in Table 4.

Completeness of Impact Sets. Section (A) of Table 4 depicts the results of the impact set detection using static, *dynamic* and hybrid analysis methods. The first column of this section displays the average, minimum and maximum sizes of the impact sets of the selected JavaScript functions, detected by TOCHAL. The second column displays the average, minimum and maximum impact set size by static analysis. The third column represents the percentage of the ratio of the impact set size of static analysis to that of TOCHAL. Similarly, the fourth and fifth columns, respectively, show the impact set size for *dynamic* analysis, and the ratio of the size of impact sets using *dynamic* analysis compared to TOCHAL. We observe an increase in completeness for all applications in favour of TOCHAL. On average, static and *dynamic* analysis methods detected 0.57 and 1.80 functions in the impact set of each sample function, respectively. The hybrid TOCHAL method, however, extracted an average of 2.07 functions to be potentially impacted by each of the sample functions.

Overall, the impact sets extracted by TOCHAL include 74% more functions on average compared to those detected by static impact analysis. The outcome conforms the findings of our earlier exploratory study, showing the prevalence and importance of DOM-related and dynamic characteristics of JavaScript in impact analysis. TOCHAL takes into account new types of entities in its dependency graph that are more aligned with the nature of JavaScript (DOM elements, XHR objects). It also recognizes new (and mostly hidden) types of relations between these entities, that lead to more complete and more precise dependency graphs and impact sets at the same time. The static analysis still remains useful in TOCHAL since *dynamic* analysis can only cover 80% of the impact sets detected by hybrid analysis and cannot replace it.

It is worth noting the small sizes of static impact sets. Considering the conservativeness of static methods, the dependency graphs in general can include many relations between functions that are not feasible in practice. Therefore, static impact sets in CIA methods for traditional languages are expected to get large and contain infeasible relations. Our results show an *opposite phenomenon* for JavaScript applications. The small sizes of static dependency graphs and the resulting impact sets attest to the difficulties and limitations of static analysis for JavaScript. The findings further confirm that new forms of definitions and usages of functions, objects, DOM elements and asynchronous objects negatively affect analysis of useful dependency graphs. Static analysis confronts more barriers during the analysis JavaScript applications that should be mitigated using an approach that supports these features.

Necessity of Hybrid Analysis. Separate data sets are collected from each type of analysis, to let us distinguish between the statically detected and dynamically invoked functions. Through these datasets, we extract the functions that were invoked dynamically but were not detected statically, and the functions that were extracted before the execution, but did not play a role at runtime. The results are summarized in section (B) of Table 4. The first column of this section contains the total number of functions that were included in our hybrid analysis. The second and third columns represent the percentages of these functions that were covered by static and *dynamic* analysis units, respectively.

The static analysis method only covers about 56% of the functions that are covered during hybrid analysis. As expected, this inadequacy is caused due to the dynamism of JavaScript even in simple function invocations. Moreover, the *dynamic* analyzer includes 86% of the functions detected by the hybrid analysis. This confirms our anticipation of incompleteness of dynamic analysis, due to its reliance on specific executed scenarios of the code. The increase in the covered JavaScript functions by our proposed hybrid analysis leads to a more

complete system dependency graph, which is used to perform the impact analysis. Hence, the proposed hybrid approach can improve the accuracy of JavaScript impact analysis.

Column 4 in section (A) of Table 4 represents the percentage of JavaScript functions that were detected by the static analyzer, but were *not* invoked during any of the executions of the applications. Note that the existence of such functions, that would be missed from pure *dynamic* analysis, is sufficient evidence for the necessity of a hybrid analysis technique. Column 5 of the same section of the table, on the other hand, displays the percentage of functions that were executed during the execution of each application, but were *not* detected by the static analyzer. The results confirm our previous intuition regarding the shortcomings of static JavaScript analysis, even in a well-established type of analysis based merely on function declarations and invocations.

7.2 Study 2: Industrial Controlled Experiment

We conducted a controlled experiment [28] in an *industrial* environment[5] to address the following two questions, derived from RQ2:

RQ2.1 Does Tochal increase the CIA task completion *accuracy*?
RQ2.2 Does Tochal decrease the CIA task completion *duration*?

Experimental Subjects

We recruited 10 participants, 5 female and 5 male, with ages ranging from 20 to 34. At the time of conducting the experiment, all participants were employed in a large software company in Vancouver. Their skills in web development ranged from medium to professional. All participants volunteered for taking part in our experiment and did not receive monetary compensation.

Experimental Design

The experiment had a "between-subject" design to avoid carryover effects. We formed two independent groups of participants. The experimental group used Tochal for performing the tasks; none of the participants were familiar with Tochal prior to attending the experimental sessions. Since Tochal is the first CIA tool for JavaScript applications, the control group performed the tasks using the development tool they used in their day-to-day web development activities (without Tochal). To avoid bias, it was important for the members of the two groups to have similar proficiency levels. We manually assigned the participants to the groups to ensure this was the case based on a pre-questionnaire evaluating their experience and expertise.

Tasks. We designed a set of four tasks that involved finding change impacts during web application maintenance activities. The tasks, outlined in Table 5, require the participants to detect and comprehend the potential impact of a change in a JavaScript function or a DOM element. Moreover, two of the tasks require participants to use their understanding of the change impact to find a bug or an inconsistency that occurs after the change.

All features of Tochal were available to the experimental group during the experimental session. We were particularly interested in assessing the usefulness of the ranking mechanism

[5] The experimental material is available at: `http://ece.ubc.ca/~saba/tochal/study-materials.zip`

Table 5 Impact analysis tasks used in the controlled experiment.

Task	Description
T1	Finding the potential impact of a DOM element (the button changing the size of the displayed slideshow images)
T2	Finding the potential impact of a JavaScript function (the function toggling the play/pause state of the slideshow)
T3	Finding a conflict after making a new change (problem in submitting new comments after changing the table containing all comments of a picture). Ranking is disabled.
T4	Finding a bug in JavaScript code (entered email format is not properly checked)

of TOCHAL, which was deployed based on our proposed impact metrics (Section 5). Hence, we designed a smaller study that enabled us to compare the effects of using the ranking. However, this comparison was only meaningful for the experimental group, who used TOCHAL and had access to its ranking feature. Having this in mind, the two debugging tasks, tasks T3 and T4 in Table 5, were designed to have similar levels of difficulty and to require similar amount of effort and expertise. However, we disabled the ranking feature for T3, while leaving it enabled for T4. These two tasks were counterbalanced to avoid order effects.

Dependent and Independent Variables. We measured two continuous variable as our dependent variables. Task completion *duration* was measured in minutes and seconds. Tasks completion *accuracy* was measured in marks based on a fixed and predefined grading rubric, and the marks were converted to percentages for consistency across all tasks.

The independent variable is a nominal variable including two levels. One level represents using TOCHAL, and the other level depicts using a different tool (e.g., Chrome DevTools, Firefox Developer Tools, or Firebug).

Experimental Object

We used Phormer [18], an online photo gallery in PHP, JavaScript, CSS and XHTML, which consisted of around 6,000 lines of code and over 38,000 downloads at the time of conducting the experiment. Throughout the experiment, the participants had to understand and debug parts of the application related to displaying a slideshow of pictures, viewing the pictures, and authoring comments.

Experimental Procedure

The experiment consisted of three main phases.

Pre-Experiment. The participants were given a pre-questionnaire form before attending the experimental session. They were required to provide information regarding their proficiency and experience in web development and software engineering. This information was used for assigning them into one of the two groups prior to the session. The pre-questionnaire also inquired about the tools the participants normally used for performing their every-day web development tasks. The answers to this question were used to determine the proficiency levels of the participants for assigning them into the control and experimental groups. The information was also used to indicate the tool the control group would use for the experiment. It is worth mentioning that all participants of the control group selected Google Chrome as their preferred web development tool. In the experimental session, the participants were introduced to TOCHAL for the first time and were trained to use it. Then they were given a few minutes to familiarize themselves with the tool, and ask us questions if needed.

Tasks. During the experimental session, the participants were asked to perform a set of tasks, as indicated in Table 5. To avoid experimental bias, each task was handed out on a separate sheet of paper to the participant, when the investigator marked the start time of the task. The investigator terminated the measurement of the task duration when the participant returned the paper to the investigator along with her answer. The task accuracy was marked after the session, using a rubric created prior to conducting the experiment.

Post-Experiment. We asked the participants to fill a post-questionnaire form after completing the tasks. The questionnaire contained questions regarding both the helpful features and the shortcomings of the tool they used in the experiment. Moreover, we enquired about the metrics our participants considered to affect the importance of an entity in the impact set.

Results and Discussions

Figures 8 and 9 depict the results of task completion duration and accuracy for both experimental and control groups.

Accuracy (RQ2.1). We ran the Shapiro-Wilk normality test on the accuracy data. The results showed that the data collected for tasks T1, T3 and T4 were not normally distributed and hence, Mann-Whitney U tests were used for analyzing the results of these tasks. The data gathered for task T2 and the total accuracy of the tasks were normally distributed and were analyzed using t-tests. The results of conducting the tests revealed a statistically significant difference for the experimental group using Tochal (Mean=84%, STDDev=14%) compared to the control group (Mean=26%, STDDev=11%); (p-value=0.0001). *Overall, participants using* Tochal *perform 223% more accurately compared to the control group, across all tasks in the experiment.*

Further, the accuracy for all tasks was higher when the participants used Tochal. The improvement was statistically significant for tasks T1, T2 and T4, but not T3. Recall from the tasks table (Table 5), that the ranking mechanism of Tochal was disabled for T3. This outcome thus emphasizes the value of ranking the impact set in helping the user find the important impacts more efficiently. Not having access to the impact ranks, participants had to expend more manual effort. Enforcing more analysis burden on the participant increases the variation in the answers based on the individual differences in abilities of the participants. The high variation prevents the statistical significance of the results of T3, in spite of the 60% higher accuracy average for Tochal users.

Duration (RQ2.2). We first used the Shapiro-Wilk test on the duration data and confirmed that the data was normally distributed. Therefore, we ran a set of t-tests on all individual tasks, as well as on the total time spent on all of the tasks throughout the experiment. The results show a statistically significant difference in the total duration for the experimental group using Tochal (Mean=19:54, STDDev=1:23), compared to the control group using other tools (Mean=35:26, STDDev=8:21); (p-value=0.01). *Overall, participants using* Tochal *spent an average of 78% less time on the same set of tasks compared to those using other tools.*

Further, the participants using Tochal showed improvements for all of the tasks compared to the control group, with the difference being statistically significant for T1. Considering the higher accuracy scores for other tasks when using Tochal, we observe that many participants of the control group terminated the tasks with incomplete and incorrect answers,

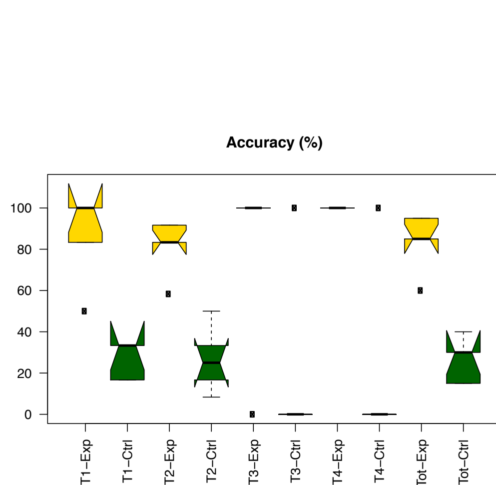


Figure 8 Task completion accuracy per task and in total for the control (ctrl) and experimental (exp) groups (RQ2.1).

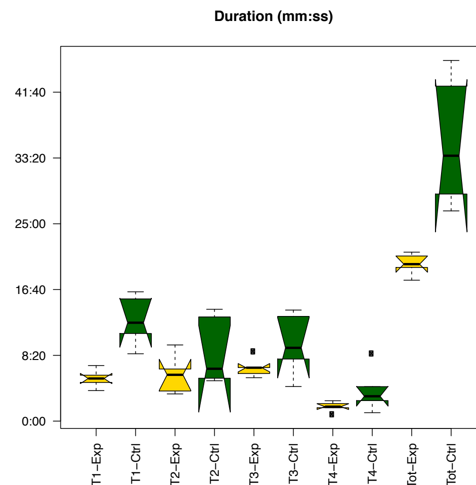


Figure 9 Task completion duration data per task and in total for the control (ctrl) and experimental (exp) groups (RQ2.2).

erroneously assuming that they had completed the task. This caused them to obtain lower accuracy scores, while still spending more time on average.

Ranking. The results of the experimental group were analyzed further to investigate the effects of using our proposed ranking system on the duration and accuracy of understanding and debugging an application after a change. T3 and T4 were both debugging tasks requiring similar levels of expertise. As mentioned earlier, the use of TOCHAL's ranking feature was disabled for T3, while it was enabled for T4. Performing a t-test on the results revealed a statistically significant difference in task completion duration between participants who used the ranking mechanism (Mean=1:50, STDDev=39), compared to those who did not (Mean=6:34, STDDev=1:16), with p-value<0.05. We used a Mann-Whitney U test to analyze the accuracy results for the ranking mechanism. Although using the ranks led to an average of 20% higher accuracy of the answers, the results were not statistically significant in this case. However, the participants completed T4 about 3.7 times faster than T3. This significant improvement highlights the importance of ranking the impact set, and is an indication of the usefulness of our impact ranking method (Section 5).

Participants' Feedback

We gathered qualitative data from both experimental and control groups through a post-questionnaire form. The questionnaire asked participants about the usefulness of the tool used in the study, its strengths, and its shortcomings. Overall, all TOCHAL users mentioned that they found the tool useful. The participants were particularly pleased with the idea of finding the potential impact of JavaScript functions and DOM elements. Understanding the dynamic behaviour and underlying dependencies were mentioned to be most useful. The users found TOCHAL to be helpful in solving the problems faster, especially in the presence of its ranking mechanism. The participants in the experimental group were also interested to see more features in TOCHAL. The feature requests were mostly attributed to improving the user interface. Some participants were also interested in having direct debugging support in the tool.

Furthermore, we asked our participants about the metrics they thought could determine the importance of an entity in the impact set. We analyzed and categorized the participants' opinions on impact metrics. A few of the final categories were considered in our ranking strategy, as we expected. These metrics included (1) distance of an entity from the change set and (2) number of dependencies of an entity. However, there were many other categories that are not included in our methods, such as: (1) number of invocations of a function, (2) visibility of the impact in the interface, (3) importance of the feature to the customers, (4) "breadth" of usage of the entity: whether it is used by multiple files, or is isolated to one file (5) complexity of the function, and (6) "history" of a function: rate of having faults in the function.

Threats to Validity

The first internal threat is the the population selection threat, and specifically the equivalency of the two groups in terms of their expertise. We addressed this threat by first manually dividing the participants into different proficiency levels, which were extracted from the information gathered in the pre-questionnaire forms. We then distributed the members of each level into control and experimental groups by random sampling. The second threat is the investigator's bias while marking the accuracy of the answers. We mitigated this threat by creating a marking rubric while designing the tasks, and using the same rubric for marking the results later. A similar threat can arise from the bias in measuring the duration of the tasks. We resolved this issue by enforcing a mutual supervision on time measurement by both the investigator and the participant. We assigned a separate sheet of paper to each task, which was handed to the participant in the beginning of the task, and was returned to the investigator after task completion. The fourth threat can be introduced by the choice of the tool that the control group used. This threat was mitigated by allowing the control group to choose the tool of their preference. Finally, the compensatory incentives were not a threat to validity as all of our participants volunteered for the experiment, with no monetary compensation.

An external threat is with regard to the representativeness of our sample of population. We mitigated this threat by recruiting professional web developers from industry as our participants. Another concern is raised regarding the representativeness of tasks used in the experiment. We used general tasks enquiring about understanding the impact of a code change and also detecting potential faults in the code, which are faced by developers in their daily professional activities.

8 Related Work

Static Analysis. Static CIA is performed by analyzing the source code without executing it. A common pattern in many traditional CIA techniques is the usage of dependency-based impact analysis methods [4]. Static impact analysis techniques typically find syntactic dependencies by performing forward slicing on the graph. This type of analysis, however, is based on assumptions made for all possible executions of the software, and hence incurs false positives, which hinders its adoption [11]. The dependency graph can become large and may contain invalid paths. Hence, the resulting impact set can be large and difficult to comprehend.

More recently, static analysis has been applied to analyze JavaScript applications. Sridharan et al. [22] adapt traditional points-to analysis for JavaScript through correlation tracking of dynamic properties in the code. Jensen et al. [12] statically model the role of

the DOM and browser in their analysis. However, they acknowledge gaps and shortcomings in their analysis, which can result in many false-positives. Feldthaus et al. [8] present an approach for constructing approximate JavaScript call graphs. However, their analysis completely ignores dynamic property accesses and interactions with the DOM. Madsen et al. [15] combine pointer analysis with use analysis to investigate the effects of JavaScript libraries and frameworks on the applications' data flow.

These techniques neglect the dynamic DOM interactions as well as event-driven, and asynchronous features of the JavaScript language. Therefore, their analysis can be incomplete for performing change impact analysis for JavaScript applications.

Dynamic Analysis. Existing dynamic methods produce a precise but incomplete analysis. Apiwattanapong et al. [2] propose a dynamic technique in which execute-after relations are used to reduce the overhead caused by the amount of collected dynamic information. Dynamic CIA tools have been applied to various fields of software engineering. For instance, Ramanathan et al. [19] avoid testing unchanged test cases by comparing strings of different traces in their tool. Chianti [20] is a CIA tool for Java that reports the change impact in terms of the subset of the test suite affected by the change. These techniques provide more precision compared to static analysis, especially when integrated with other techniques such as information retrieval [6][9]. However, they do not target JavaScript code and its unique analysis challenges, such as DOM interactions, dynamic function calls involving event propagations, and asynchronous callbacks.

Wei and Ryder's recent JavaScript blended analysis approach [26] and state-sensitive points-to analysis [27] are perhaps the closest to our work. Their work integrates the information gathered during both static and dynamic analyses to perform a points-to analysis of JavaScript applications. However, their methods do not focus on analyzing the change impact, and hence do not incorporate the dependencies that are formed through DOM interactions and asynchronous JavaScript mechanisms. Moreover, they do not take into account the important role of events and event propagation [25] on the DOM tree, which connects JavaScript functions, unlike our analysis which does.

9 Concluding Remarks

The dynamic, asynchronous, and event-based nature of JavaScript and its interactions with the DOM make modern web applications highly interactive and responsive to users. These same features also introduce new types of dependencies into the system, making the prediction and detection of change impact challenging for developers. In this paper, we proposed an automated technique, called TOCHAL, for performing a hybrid DOM-sensitive change impact analysis for JavaScript. TOCHAL builds a novel hybrid system dependency graph, by inferring and combing static and dynamic call graphs. Our technique ranks the detected impact set based on the relative importance of the entities in the hybrid graph. Our evaluation shows that the dynamic and DOM-based JavaScript features occur in real applications and can lead to significant means of impact propagation. Furthermore, we find that a hybrid approach leads to a more complete analysis compared with a pure static or dynamic analysis. Finally, our industrial controlled experiment shows that TOCHAL increases developers' performance, by helping them to perform maintenance tasks faster and more accurately. In future work, we plan to extend the granularity of the analysis to intra-procedural JavaScript dependencies and explore more effective ranking functions.

References

1. Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. Understanding JavaScript event-based interactions. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 367–377. ACM, 2014.
2. Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 432–441. ACM, 2005.
3. Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society, 1996.
4. Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.
5. Ben Breech, Mike Tegtmeyer, and Lori Pollock. Integrating influence mechanisms into impact analysis for increased precision. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 55–65. IEEE, 2006.
6. Bogdan Dit, Michael Wagner, Shasha Wen, Weilin Wang, Mario Linares-Vásquez, Denys Poshyvanyk, and Huzefa Kagdi. Impactminer: A tool for change impact analysis. In *Companion Proceedings of the International Conference on Software Engineering (ICSE)*, pages 540–543. ACM, 2014.
7. Document Object Model (DOM). `http://www.w3.org/DOM/`.
8. Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013.
9. M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 430–440. ACM, 2012.
10. Google. DevTools: The Chrome Developer Tools. https://developer.chrome.com/devtools.
11. Lile Hattori, Dalton Guerrero, Jorge Figueiredo, Joao Brunet, and Jemerson Damásio. On the precision and accuracy of impact analysis techniques. In *Proceedings of the International Conference on Computer and Information Science*, pages 513–518, 2008.
12. Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 59–69. ACM, 2011.
13. Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.
14. 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. *Communications of the ACM*, 58(2):44–46, 2015.
15. Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 499–509. ACM, 2013.
16. Mutation summary. `https://code.google.com/p/mutation-summary/`.
17. Maksym Petrenko and Václav Rajlich. Variable granularity for improving precision of impact analysis. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 10–19. IEEE, 2009.
18. Phormer PHP photo gallery. `http://p.horm.org/er/`.
19. Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 241–252. IEEE, 2006.

20 Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of Java programs. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 432–448. ACM, 2004.

21 Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, 2010.

22 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 435–458. Springer, 2012.

23 T. J. Watson Libraries for Analysis. WALA. http://wala.sourceforge.net/.

24 Tochal. `https://github.com/saltlab/tochal`.

25 W3C. Document Object Model (DOM) level 2 events specification. http://www.w3.org/TR/DOM-Level-2-Events/, 13 November 2000.

26 Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for JavaScript. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 336–346. ACM, 2013.

27 Shiyi Wei and Barbara G Ryder. State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 1–26. Springer, 2014.

28 Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction.* Kluwer, 2000.

Intensional Effect Polymorphism*

Yuheng Long[1], Yu David Liu[2], and Hridesh Rajan[3]

1,3 Iowa State University
226 Atanasoff Hall, Ames, IA 50011, USA
{csgzlong,hridesh}@iastate.edu
2 SUNY Binghamton
Binghamton NY 13902, USA
davidL@cs.binghamton.edu

Abstract

Type-and-effect systems are a powerful tool for program construction and verification. We describe *intensional effect polymorphism*, a new foundation for effect systems that integrates static and dynamic effect checking. Our system allows the effect of polymorphic code to be intensionally inspected through a lightweight notion of dynamic typing. When coupled with parametric polymorphism, the powerful system utilizes runtime information to enable precise effect reasoning, while at the same time retains strong type safety guarantees. We build our ideas on top of an imperative core calculus with regions. The technical innovations of our design include a *relational* notion of effect checking, the use of *bounded existential types* to capture the subtle interactions between static typing and dynamic typing, and a *differential alignment* strategy to achieve efficiency in dynamic typing. We demonstrate the applications of intensional effect polymorphism in concurrent programming, security, graphical user interface access, and memoization.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, F.3.2 Semantics of Programming Languages

Keywords and phrases intensional effect polymorphism, type system, hybrid typing

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.346

1 Introduction

In a type-and-effect system [27, 36], the type information of expression e encodes and approximates the computational effects σ of e, such as how memory locations are accessed in e. Type-and-effect systems – or effect systems for short in this paper – have broad applications (*e.g.*, [2, 29, 24, 6]). Improving their expressiveness and precision through static approaches is a thoroughly explored topic, where many classic language design (*e.g.*, [22, 15, 35, 4, 26]) and program analysis (*e.g.*, [33, 3]) techniques may be useful.

Purely static effect systems are a worthy direction, but looking forward, we believe that a complementary *foundation* is also warranted, where the default is a system that can fully account for and exploit runtime information, aided by static approaches for optimization. Our belief is shaped by two insights. First, emerging software systems increasingly rely on dynamic language features: reflection, dynamic linking/loading, native code interface, flexible meta programming in script languages, to name a few. Second, traditional hurdles

* This work was partially supported by the US National Science Foundation (NSF) under grants CCF-14-23370, CCF-13-49153, CCF-11-17937, CCF-10-17334, CNS-07-09217, CNS-06-27354, and CAREER awards CCF-08-46059 and CCF-10-54515.

© Yuheng Long and Yu David Liu, and Hridesh Rajan;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 346–370

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

defying precise static reasoning – such as expression ordering, branching, recursion, and object dynamic dispatch – are often amplified in the context of effect reasoning.

In this paper, we develop *intensional effect polymorphism*, a system that integrates static and dynamic effect reasoning. The system relies on dynamic typing to compensate for the conservativeness of traditional static approaches and account for emerging dynamic features, while at the same time harvesting the power of static typing to vouchsafe for programs whose type safety is fundamentally dependent on runtime decision making. Consider the following example:

▶ **Example 1** (Conservativeness of Static Typing for Race-Free Parallelism)**.** Imagine we would like to design a type system to guarantee race freedom of parallel programs. Let expression $\mathrm{e}||\mathrm{e}'$ denote running e and e' in parallel, whose typing rule requires that e and e' have memory access effects over disjoint regions. Further, let `r1` and `r2` be disjoint regions. The following program is race-free, even though a purely static effect system is likely to reject it:

```
1                        (λx.λy. (x := 1) || !y)
2                                 (if 1 > 0 then ref_r1 0 else ref_r2 0)
3                                 (if 0 > 1 then ref_r1 0 else ref_r2 0)
```

Observe that parametric polymorphism is not helpful here: `x` and `y` can certainly be typed as region-polymorphic, but the program remains untypable. The root cause of this problem is that race freedom only depends on the *runtime* behaviors of `(x:=1) || !y`, which only depends on what `x` and `y` are *at runtime*.

Inspired by Harper and Morrisett [19], we propose an effect system where polymorphic code may intensionally inspect effects at runtime. Specifically, expression **assuming** e $\mathbb{R}$ e' **do** e_1 **else** e_2 inspects whether the runtime (effect) type of e and that of e' satisfy binary relation $\mathbb{R}$, and evaluates e_1 if so, or e_2 otherwise. Our core calculus leaves predicate $\mathbb{R}$ abstract, which under different instantiations can support a family of concrete type-and-effect language systems. To illustrate the example of race freedom, let us consider $\mathbb{R}$ being implemented as region disjointness relation #. The previous example can be written in our calculus as follows.

▶ **Example 2** (Intensional Effect Polymorphism for Race-Free Parallelism)**.** The following program type checks, with the static system and the dynamic system interacting in interesting ways:

```
1                  (λx.λy. assuming (x := 0) # !y do (x := 1) || y)
2                           (if 1 > 0 then ref_r1 0 else ref_r2 0)
3                           (if 0 > 1 then ref_r1 0 else ref_r2 0)
```

Static typing can guarantee that the lambda abstraction in the first line is well-typed regardless of how it is applied, good news for modularity. Dynamic typing provides precise typing for expression `(x := 0)` and expression `!y` – exploiting the runtime type information of `x` and `y` – allowing for a more precise disjointness check. Observe that in our calculus, the subexpression participating in the **assuming** check – `x := 0` here – does not have to be syntactically isomorphic to the expression in the **do** subexpression, `x := 1`. For simplicity, we omit the **else** expression in this example.

Technical Innovations

On the highest level, our system shares the philosophy with a number of type system designs hybridizing static checking and dynamic checking (*e.g.*, [14, 34, 18]), and some in the contexts of effect reasoning [5, 20]. To the best of our knowledge however, this is the first time intensional type analysis is applied to effect reasoning. This combination is powerful,

because not only effect reasoning can rely on runtime type information, but also parametric polymorphism is fully retained. For example, observe that in the example above, the types for `x` and `y` are parametric, not just "unknowns" or "dynamic". Let us look at another example:

▶ **Example 3** (Parametric Polymorphism Preservation)**.** For the following program, the parallel execution in the second line is statically guaranteed to be type-safe in our system. Programs written with intensional effect polymorphism do not have runtime type errors.

```
1    let s = λx.λy. assuming (x := 0) # !y do (x := 1) || !y in
2             (s ref_r1 0 ref_r2 0) || (s ref_r3 0 ref_r4 0)
```

In addition, intensional effect polymorphism goes beyond a mechanical adaptation of Harper-Morrisett, with several technical innovations we now summarize. The most remarkable difference is that the intensionality of our type system is enabled through *dynamic typing*. At runtime, the evaluation of expression **assuming** e $\mathbb{R}$ e′ **do** e_1 leads to the dynamic typing of e and e′. In contrast, the classic intensional type analysis performs a `typecase`-like inspection on the runtime instantiation of the polymorphic type. Our strategy is more general, in that it not only subsumes the former – indeed, a type derivation conceptually constructed at runtime must have leaf nodes as instances of value typing – but also allows (the effect of) arbitrary expressions to be inspected at runtime. We believe this design is particularly relevant for effect reasoning, because it has less to do with the effect of polymorphically-typed variables, and more with *where* these variables appear in the program.

Second, we design the runtime type inspection through a *relational* check. In the **assuming** expression, our system dynamically checks whether $\mathbb{R}$ holds, instead of computing what the effect of e or e′ is. The relational design does not require programmers to explicitly provide an "effect specification/pattern" of the runtime type – a task potentially daunting as it may either involve enumerating region names, or expressing conditional specifications such as "a region that some other expression does not touch." Many safety properties reasoned about by effect systems are relational in nature, such as thread interference.

Third, the subtle interaction between static typing and dynamic typing poses a unique challenge on type soundness in the presence of effect subsumption. We elaborate on this issue in Section 4.4. We introduce a notion of *bounded existential types* to differentiate but relate the types assumed by the static system and those by the dynamic system.

Finally, a full-fledged construction of type derivations at runtime for dynamic typing would incur significant overhead. We design a novel optimization to allow for efficient runtime effect computation, eliminating the need for dynamic derivation construction while producing the same result. The key insight is that we could align the static type derivation and the (would-be-constructed) dynamic type derivation, and compute the effects of the latter simply by substituting the difference of the two, a strategy we call *differential alignment*. We will detail this design in Section 5.

We formalize intensional effect polymorphism in λ_{ie}, an imperative call-by-value λ-calculus with regions. In summary, this paper makes the following contributions:

- It describes a hybrid type system for effect reasoning centering on intensional polymorphism.
- It develops a sound type system and operational semantics where relational effect inspection is made abstract.
- It illuminates the subtleties resulting from the difference between static effect reasoning and dynamic effect reasoning, and proposes bounded existential types to preserve soundness, and differential alignment to promote efficiency.
- It demonstrates the impact of our design by extending the core calculus to the applications of safe parallelism support, security, UI access, and memoization.

```
1  class Pair {
2    int ft = 1, sd = 2;

4    int applyTwice(Op f) {
5      assuming ft = f.op(0)   #   sd = f.op(5)
6      do ft = f.op(f.op(ft)) || sd = f.op(f.op(sd));
7    else ft = f.op(f.op(ft)) ;   sd = f.op(f.op(sd));
8    }
9  }

11 Pair pr = new Pair();
12 Op o = (Op) newInstance(readFile("filePath"));
13 pr.applyTwice(o);
```

```
14 interface Op { int op(int i); }

16 class Pref implements Op {
17   int sum = 0;?
18   // effect:  write sum
19   int op(int i) { sum += i; }
20 }

22 class Hash implements Op {
23   // effect:  pure, no effect
24   int op(int i) { hash(i); }
25 }
```

Figure 1 An application of intensional effects in enforcing safe parallelism.

2 Motivating Examples

In this section, we demonstrate the applicability of intensional effects in reasoning about safe parallelism, information security, consistent UI access, and program optimization. In each of these applications, the refined notion of type safety is fundamentally dependent on runtime decision making, *i.e.*, whether the relation $\mathbb{R}$ is satisfied. We instantiate the effect relation operator $\mathbb{R}$ with different concrete relations between effects of expressions.

As in previous work [17, 8], we optionally extend standard Java-like syntax with *region declarations* when the client language deems them necessary. In that case, a variable declaration may contain both type and region annotations, *e.g.*, `JLabel j` **in** `ui` declares a variable `j` in region `ui`. For client languages where regions are not explicitly annotated, different abstract locations (such as different fields of an object) are treated as separate regions.

2.1 Safe Parallelism

We demonstrate the application of intensional effects in supporting safe parallelism, where safety in this context refers to the conventional notion of thread non-interference (race freedom) [27]. Concretely, Figure 1 is a simplified example of "operation-agnostic" data parallelism, where the programmer's intention is to apply some statically unknown operation (encapsulated in an `Op` object) – here implemented through reflection – to a data set, here simplified as a pair of data `ft` and `sd`. The programmer wishes to "best effort" leverage parallelism to process `ft` and `sd` in parallel, without sacrificing thread non-interference. The tricky problem of this notion of safety is it depends on what `Op` object is. For instance, parallel processing of the pair with the `Hash` object is safe, but not when the operation at concern is the prefix sum operator [7], encapsulated as `Pref`.

Static reasoning about the correctness of the parallel composition could be challenging in this example, because the `Op` object remains unknown until `applyTwice` is invoked at runtime.

The **assuming** expression (line 5) helps the program retain strong type safety guarantees for parallel composition (line 6), while utilizing the runtime information to enable precise reasoning. At runtime, the **assuming** expression intensionally inspects the effects of the expressions `ft = f.op(0)` and `sd = f.op(5)`. If they satisfy the binary relation #, parallelism will be enabled. If `f` points to a `Hash` object, the # relation will be true and the program enjoys safe concurrency (line 6). On the other hand, if `f` points to a `Pref` object, the program will be run sequentially, desirable for race freedom safety.

```
1  class Page {
2    String searchBox = "";
3    String url = "wsj.com/search?";
4    String location = "";

6    String load_adv(ThirdParty adv) {
7      assuming url ◊ adv.show(this)
8      do exec url adv.show(this);
9      else "no advertisement";
10   }

12   int search(ThirdParty adv) {
13     load_adv(adv);
14     location = url + searchBox;
15   }
16 }
```

```
17 interface ThirdParty { String show(Page p); }

19 class Good implements ThirdParty {
20   String show(Page p) { "404"; }
21 }

23 class Evil implements ThirdParty {
24   String show(Page p) {
25     p.url = "evil.com";
26   }
27 }

29 ThirdParty adv = (ThirdParty)
30     newInstance(readFile("filePath"));
31 new Page().render(adv);
```

Figure 2 An application of intensional effects in preventing security vulnerabilities.

2.2 Information Security

As another application of intensional effects, consider its usage in preventing security vulnerabilities. Figure 2 presents an adapted (`wsj.com`) example of real-world security vulnerabilities [11]. The `Page` class allows users to search information within the site. Once the `search` is called, the page will redirect to a web page corresponding to the `url` and `searchBox` strings (the redirection is represented as changing the `location` variable for simplicity). The page, when created, inserts a third party advertisement (line 8).

The third party code can be malicious, *e.g.*, it can modify the search `url` and redirects the search to a malicious site, from which the whole system could be compromised, *e.g.*, the `Evil` third-party code. Ensuring the key security properties becomes challenging with the dynamic features because the third party code is only available at runtime, loaded using reflection. The expression **exec** e_1 e_2 (line 7) encodes a *check-then-act* programming pattern. It executes e_2 only if it does not read nor write any object accessible by e_1 and otherwise it gets stuck.

With intensional effects, users can intensionally inspect a third party code e whenever e is dynamically loaded. The intensional inspection, accompanied with a relational policy check, ensures that e does not access any sensitive data (the `url`), specified using the relation $\Diamond$. It also ensures that the **exec** expression does not get stuck.

2.3 Consistent Graphical User Interface (GUI) Access

We show how intensional effects can be used to reason about the correctness of a GUI usage pattern, common in Subclipse, JDK, Eclipse and JFace [16]. Typically, GUI has a single *UI thread* handling events in the "event loop". This UI thread often spawns separate *background threads* to handle time-consuming operations. Many frameworks enforce a single-threaded GUI policy: only the UI thread can access the GUI objects [16]. If this policy is violated, the whole application may abort or crash. Figure 3 shows a simplified example of a UI thread that pulls an event from the `eventloop` and handles it. In the application, all UI elements reside in the `ui` region (declared at line 14), *e.g.*, the field `j` at line 23.

The safety here refers to no UI access in any background thread. The tricky problem here is that the events arrive at runtime with different event handlers. Some handlers may access UI objects while the others do not. Therefore, the correctness of spawning a thread to handle a new event, depends heavily on what objects the corresponding event handler has. For instance, the handler containing a `NonUI` object can be executed in a background thread, while `UIAccess` should not. The expression **spawn** e_1 e_2, executes e_2 in a background thread

```
1  class UIThread {
2    JLabel global in ui = new JLabel();
3    void eventloop(Runnable closure) {
4      assuming global ∅ closure.run()
5        do spawn global closure.run();
6        else closure.run();
7    }
8  }

10 Runnable closure;
11 if (1 > 0) closure = new NonUI();
12 else closure = new UIAccess();
13 new UIThread().eventloop(closure);

14 region ui;

16 interface Runnable { String run(); }

18 class NonUI implements Runnable {
19   String run() { "does nothing"; }
20 }

22 class UIAccess implements Runnable {
23   JLabel j in ui = new JLabel();
24   String run() { j.val = "UI"; }
25 }
```

Figure 3 An application of intensional effects in disciplining UI access.

```
1  class Mem {
2    Integer input = new Integer();

4    int comp(Mutate m, Integer x) {
5      int cache = heavy(input);
6      assuming m.mutate(x) ♮ heavy(input)
7      do lookup m.mutate(x) heavy(input); cache
8      else m.mutate(x); heavy(input);
9    }

11   int heavy(Integer i) { /* ... */}
12 }

13 class Integer { int i = 0; }

15 class Mutate {
16   int mutate(Integer input) {
17     input.i = 101;
18   }
19 }

21 Memo mm = new Mem();
22 Mutate mu = new Mutate();
23 if (1 > 0) mm.comp(mu, mm.input);
24 else mm.comp(mu, new Integer());
```

Figure 4 An application of intensional effects in providing effective memoization.

only if it does not allocate, read or write any object in the region accessed by e_1. Otherwise it gets stuck.

The **assuming** expression, used by the UI thread, statically guarantees strong type safety for the **spawn** expression, so it does not get stuck. It also utilizes precise runtime information to distinguish handlers with no UI accesses from other handlers. If a handler satisfies relation $\varnothing$, it can be safely executed by a background thread. The relation $\varnothing$ is satisfied if the RHS expression does not allocate, read/write any region denoted by the LHS expression.

2.4 Program Optimization – Memoization

As another application, we utilize intensional effects to implement a proof-of-concept memoization technique in a sequential program. Memoization is an optimization technique where the results of expensive function calls are cached and these cached results are returned when the inputs and the environment of the function are the same.

Figure 4 presents a simplified application where repeated tasks, here the `heavy` method calls on line 5 and 8, are performed. These two tasks are separated by a small computation `mutate`, forming a *compute-mutate* pattern [9]. We leave the body of the method `heavy` intentionally unspecified, which could represent a set of computationally expensive operations. It could, *e.g.*, generate the power set `ps` of a set of `input` elements and return the size of `ps` or do the Bogosort.

The second `heavy` task needs not be recomputed in full if the `mutate` invocation does not modify the input nor the environment of `heavy`. If so, the cached result of the first call can be reused and the repeated computation can be avoided. The expression **lookup** e_1 e_2 executes the expressions e_1 only if e_1 does not write to objects in the regions read by e_2. Otherwise it gets stuck.

Ensuring that the **lookup** expression does not get stuck is challenging. This is because the validity of the cached result depends on the runtime value of both the mutation `m` and

$$
\begin{array}{lll}
v & ::= \mathrm{b} \mid \lambda \mathrm{x}:\mathrm{T.e} & \textit{value} \\
\mathrm{e} & ::= v \mid \mathrm{x} \mid \mathrm{e}\ \mathrm{e} \mid \textbf{let}\ \mathrm{x} = \mathrm{e}\ \textbf{in}\ \mathrm{e} \mid \textbf{if}\ \mathrm{e}\ \textbf{then}\ \mathrm{e}\ \textbf{else}\ \mathrm{e} & \textit{expression} \\
 & \quad \mid \textbf{ref}\ \rho\ \mathrm{T}\ \mathrm{e} \mid !\mathrm{e} \mid \mathrm{e} := \mathrm{e} & \\
 & \quad \mid \textbf{assuming}\ \mathrm{e}\ \mathbb{R}\ \mathrm{e}\ \textbf{do}\ \mathrm{e}\ \textbf{else}\ \mathrm{e} \mid \mathbb{SAFE}\ \mathrm{e}\ \mathrm{e} & \\
\mathrm{T} & ::= \alpha \mid \textbf{Bool} \mid \mathrm{T} \xrightarrow{\sigma} \mathrm{T}' \mid \textbf{Ref}_\rho\ \mathrm{T} & \textit{type} \\
\rho & ::= \overline{\zeta} & \textit{region} \\
\zeta & ::= \mathrm{r} \mid \gamma & \textit{region element} \\
\sigma & ::= \overline{\omega} & \textit{effect} \\
\omega & ::= \varsigma \mid acc_\rho \mathrm{T} & \textit{effect element} \\
acc & ::= \textbf{init} \mid \textbf{rd} \mid \textbf{wr} & \textit{access right}
\end{array}
$$

Figure 5 λ_{ie} Abstract Syntax. (Throughout the paper, notation $\overline{\bullet}$ represents a set of $\bullet$ elements, and notation $\vec{\bullet}$ represents a sequence of $\bullet$ elements.)

its input `x`. For example, if the parameter `x` is a new object as the one created at line 24, the cache is valid, while the one represented by `mm.input` (line 23) is not valid.

The **assuming** expression solves the problem: the safety of the **lookup** expression is statically guaranteed. At runtime, with precise dynamic information, the intensional binary $\natural$ relational check ensures that the write accesses of the LHS do not affect the RHS expression. If this relation is satisfied, the cache is valid and can be reused.

Other optimizations

Intensional effect polymorphism can be used for other similar optimizations, *e.g.*, record-and-reply style memoization, common sub-expression elimination, loop-invariant code motion and redundant load elimination. In all these applications, if the mutation, similar to the style of, for example, `m.mutate(x)`, is infrequent or does not modify a large portion of the heap, the cached results can avoid repeated expensive computations.

Summary

The essence of intensional effect polymorphism lies in the *interesting interplay between static typing and dynamic typing*. Static typing guarantees that the potentially unsafe expressions are only used under runtime “safe” contexts (*i.e.*, those that pass the relational effect inspection), in highly dynamic scenarios such as parallel composition, loading third party code, handling I/O events, and data reuse. Dynamic typing exploits program runtime type information to allow for more precise effect reasoning, in that “safe” contexts can be dynamically decided upon based on runtime type/effect information.

3 λ_{ie} Abstract Syntax

To highlight the foundational nature of intensional effect polymorphism, we build our ideas on top of an imperative region-based lambda calculus. The abstract syntax of λ_{ie} is defined in Figure 5. Expressions are standard for an imperative λ calculus, except the last two forms which we will soon elaborate. We do not model integers and unit values, even though our examples may freely use them. Since **if** e **then** e **else** e plays a non-trivial role in our examples, we choose to model it explicitly. As a result, boolean values $\mathrm{b} \in \{\textit{true}, \textit{false}\}$ are also explicitly modeled. Metavariable x represents variables.

Our core syntax is expressive enough to encode the core program logic of the examples in Section 2. However, it does not model objects for simplicity without the loss of generality. An extension with objects and classes is mostly standard [30] and is included in our technical report [25].

We introduced expression **assuming** $\mathrm{e}\ \mathbb{R}\ \mathrm{e}'$ **do** e_0 **else** e_1, where from now on we call e and e' the condition expressions, e_0 the **do** expression, and e_1 the **else** expression. At runtime, this expression retrieves the effects of e and e' through dynamic typing, without evaluating e or e'. The timing of gaining this knowledge is important: the conditions will not be evaluated and the **do** expression is not evaluated yet. In other words, even though our system relies on runtime information, it is not an *a posteriori* effect monitoring system.

A General Framework

As illustrated in Section 2, effect reasoning has diverse applications. We aimed to design a *general* framework for effect reasoning, which can be concretized to different "client" languages. To achieve this goal, we choose to (1) leave the definition of the binary relation $\mathbb{R}$ abstract; (2) include an abstract $\mathbb{SAFE}$ e e' expression, which is type-safe iff $\mathrm{e}\ \mathbb{R}\ \mathrm{e}'$ holds. The $\mathbb{R}$ relation and the $\mathbb{SAFE}$ expression can be concretized to different "client" languages to capture different application domain goals. For example, when $\mathbb{R}$ is concretized to thread non-interference, one possible concretization of $\mathbb{SAFE}$ e e' is parallel composition $\mathrm{e}||\mathrm{e}'$. The instantiations of $\mathbb{R}$ of the applications in Section 2 are shown in Figure 6. The only requirement for $\mathbb{R}$ is that it must be monotone [28, 8], *i.e.*, closed under effect subsetting. Concretely, it says that if $\sigma\ \mathbb{R}\ \sigma'$, $\sigma_0 \subseteq \sigma$ and $\sigma_1 \subseteq \sigma'$, then $\sigma_0\ \mathbb{R}\ \sigma_1$. All instantiations in Section 2 satisfy this requirement.

Types, Regions, and Effects

Programmer types are either primitive types, reference types $\mathbf{Ref}_\rho$ T for store values of type T in region ρ, or function types $\mathrm{T} \xrightarrow{\sigma} \mathrm{T}'$, from T to T' with σ as the effect of the function body. Last but not least, as a framework with parametric polymorphism, types may be type variables α.

Our notion of regions is standard [36, 27], an abstract collection of memory locations. A region in our language can either be demarcated as a constant r, or parametrically as a region variable γ.

An effect is a set of effect elements, either an effect variable ς, or $acc_\rho\mathrm{T}$, representing an access *acc* to region ρ whose stored values are of type T. Access rights **init**, **rd**, **wr** represent allocation, read, and write, respectively.

As the grammar suggests, our framework is a flexible system where a type, a region, or an effect may all be parametrically polymorphic.

4 The Type System

This section describes the static semantics to our type-and-effect system. Overall, the type system associates each expression with effects, a goal shared by all effect systems. The highlight is how to construct a *precise* and *sound* effect system to support dynamic-typing-based intensionality. The precision of this type system is rooted at the $\mathbb{R}$ relation enforcement, at **assuming** time, based on effects computed by dynamic typing over runtime values and their types. Our static type system is designed so that any $\mathbb{SAFE}$ expression appearing in the **do** branch does not need to resort to runtime enforcement and the $\mathbb{R}$ relation is guaranteed

Safe Parallel Composition, Section 2.1

$\mathrm{e}\ \mathbb{R}\ \mathrm{e}' \stackrel{\text{def}}{=} \mathrm{e}\#\mathrm{e}'$ "Two effects do not interfere."

$\mathbb{SAFE}\ \mathrm{e}\ \mathrm{e}' \stackrel{\text{def}}{=} (\mathrm{e} \mid\mid \mathrm{e}')$ "Run the two expressions in parallel."

$\#$ is defined as:

$$\emptyset \# \sigma \qquad \frac{\sigma \# \sigma'' \quad \sigma' \# \sigma''}{\sigma \cup \sigma' \# \sigma''} \qquad \frac{\sigma' \# \sigma}{\sigma \# \sigma'} \qquad \mathbf{rd}_\rho \mathrm{T} \# \mathbf{rd}_{\rho'} \mathrm{T}' \qquad \frac{\rho \neq \rho'}{\mathbf{rd}_\rho \mathrm{T} \# \mathbf{wr}_{\rho'} \mathrm{T}'} \qquad \frac{\rho \neq \rho'}{\mathbf{wr}_\rho \mathrm{T} \# \mathbf{wr}_{\rho'} \mathrm{T}'}$$

Information Security, Section 2.2

$\mathrm{e}\ \mathbb{R}\ \mathrm{e}' \stackrel{\text{def}}{=} \mathrm{e}\lozenge\mathrm{e}'$ "Expression e′ does not read/write regions accessible by e."

$\mathbb{SAFE}\ \mathrm{e}\ \mathrm{e}' \stackrel{\text{def}}{=} \mathbf{exec}\ \mathrm{e}\ \mathrm{e}'$ "Execute e′ if it does not read/write the regions by e."

$\lozenge$ is defined as:

$$\sigma \lozenge \emptyset \qquad \frac{\sigma \lozenge \sigma'' \qquad \sigma' \lozenge \sigma''}{\sigma \cup \sigma' \lozenge \sigma''} \qquad \frac{\sigma'' \lozenge \sigma \qquad \sigma'' \lozenge \sigma'}{\sigma'' \lozenge \sigma \cup \sigma'} \qquad \frac{\rho \neq \rho'}{acc_\rho \mathrm{T} \lozenge \mathbf{rd}_{\rho'} \mathrm{T}'} \qquad \frac{\rho \neq \rho'}{acc_\rho \mathrm{T} \lozenge \mathbf{wr}_{\rho'} \mathrm{T}'}$$

UI Access, Section 2.3

$\mathrm{e}\ \mathbb{R}\ \mathrm{e}' \stackrel{\text{def}}{=} \mathrm{e}\varnothing\mathrm{e}'$ "Expression e′ does not access regions accessible by e."

$\mathbb{SAFE}\ \mathrm{e}\ \mathrm{e}' \stackrel{\text{def}}{=} \mathbf{spawn}\ \mathrm{e}\ \mathrm{e}'$ "Execute e′ in another thread if it accesses no region by e."

$\varnothing$ is defined as:

$$\sigma \varnothing \emptyset \qquad \frac{\sigma'' \varnothing \sigma \qquad \sigma'' \varnothing \sigma'}{\sigma'' \varnothing \sigma \cup \sigma'} \qquad \frac{\sigma \varnothing \sigma'' \qquad \sigma' \varnothing \sigma''}{\sigma \cup \sigma' \varnothing \sigma''} \qquad \frac{\rho \neq \rho'}{acc_\rho \mathrm{T} \varnothing acc_{\rho'} \mathrm{T}'}$$

Memoization, Section 2.4

$\mathrm{e}\ \mathbb{R}\ \mathrm{e}' \stackrel{\text{def}}{=} \mathrm{e}\natural\mathrm{e}'$ "RHS's read has no dependcy on the LHS's write"

$\mathbb{SAFE}\ \mathrm{e}\ \mathrm{e}' \stackrel{\text{def}}{=} \mathbf{lookup}\ \mathrm{e}\ \mathrm{e}'$ "Execute e if e writes no region read by e′."

$\natural$ is defined as:

$$\emptyset \natural \sigma \qquad \frac{\sigma \natural \sigma'' \qquad \sigma' \natural \sigma''}{\sigma \cup \sigma' \natural \sigma''} \qquad \mathbf{rd}_\rho \mathrm{T} \natural \sigma \qquad \sigma \natural \mathbf{wr}_\rho \mathrm{T} \qquad \frac{\rho \neq \rho'}{\mathbf{wr}_\rho \mathrm{T} \natural \mathbf{rd}_{\rho'} \mathrm{T}'}$$

Figure 6 Client Implementation of $\mathbb{R}$ and $\mathbb{SAFE}$ e e.

$$
\begin{array}{llll}
\Gamma & ::= & \overrightarrow{\mathrm{x} \mapsto \tau} & \textit{type environment} \\
\tau & ::= & \forall \vec{g}.\exists\Sigma.\mathrm{T} & \textit{type scheme} \\
g & ::= & \alpha \mid \gamma \mid \varsigma & \textit{generic variable} \\
gs & ::= & \mathrm{T} \mid \rho \mid \sigma & \textit{generic structure} \\
\Phi & ::= & \overline{\Lambda} & \textit{relationship set} \\
\Sigma & ::= & \overline{g \preceq: gs} & \textit{subsumption set} \\
\Lambda & ::= & \sigma \;\mathbb{R}\; \sigma \mid \forall \vec{g}.\Sigma & \textit{relationship}
\end{array}
$$

Figure 7 λ_{ie} Type System Definitions.

to hold by the static type system. As we shall see, this leads to non-trivial challenges to soundness, as static typing and dynamic typing make related – yet different – assumptions on effects.

4.1 Definitions

Relevant structures of our type system are defined in Figure 7.

Type Environment and Type Scheme

Type environment Γ maps variables to type schemes, and we use notation $\Gamma(x)$ to refer to T where the rightmost occurrence of $\mathrm{x} : \mathrm{T}'$ for any T' in Γ is $\mathrm{x} : \mathrm{T}$.

A type scheme is similar to the standard notion where names may be bound through quantification [13]. Our type scheme, in the form of $\forall \vec{g}.\exists\Sigma.\mathrm{T}$, supports both universal quantification and existential quantification. Our use of universal quantification is mundane: the same is routinely used for parametric polymorphism systems. Observe that in our system, type variables, region variables, and effect variables may all be quantified, and we use a metavariable g for this general form, and call it a *generic variable*. Similarly, we use a unified variable gs to represent either a type, a region, or an effect, and call it a *generic structure* for convenience. Existential quantification is introduced to maintain soundness, a topic we will elaborate in a later subsection. For now, only observe that existentially quantified variables appear in the type scheme as a sequence of $g \preceq: gs$, each of which we call a *subsumption relationship*. Here we also informally say g is existentially quantified, with *bound* gs. When $\vec{g}$ is a sequence of 0 and Σ is empty, we also shorten the type scheme $\forall \vec{g}.\exists\Sigma.\mathrm{T}$ as T. Type schemes are alpha-equivalent.

Relationship Set

Another crucial structure to construct our type system is the *relationship set* Φ. On the high level, this structure captures the relationships between generic structures. Concretely, it is represented as a set whose element may either be an *abstract effect relationship* $\sigma \;\mathbb{R}\; \sigma'$ – denoting two effects σ and σ' conform to the $\mathbb{R}$ relation – or a *subsumption context relationship*. The latter is represented as $\forall \vec{g}.\Sigma$. Intuitively, a subsumption context relationship is a collection of subsumption relationships, except some of its generic variables may be universally quantified. Subsumption context relationships are alpha-equivalent.

As we shall see, the relationship set plays a pivotal role during type checking. At each step of derivation, this structure represents what one can assume about effects. For example, the interplay between **assuming** and $\mathbb{SAFE}$ is represented through whether the relationship set constructed through typing **assuming** can entail the relationship that makes the $\mathbb{SAFE}$ expression type-safe. Our relationship set may have a distinct structure, but effect system

Subtyping: $\Phi \vdash \mathrm{T} \preceq: \mathrm{T'}$

$$\text{(TYPE-REFL)}\ \Phi \vdash \mathrm{T} \preceq: \mathrm{T} \qquad \text{(TYPE-TRAN)}\ \frac{\Phi \vdash \mathrm{T} \preceq: \mathrm{T}_0 \quad \Phi \vdash \mathrm{T}_0 \preceq: \mathrm{T}'}{\Phi \vdash \mathrm{T} \preceq: \mathrm{T}'} \qquad \text{(TYPE-REF)}\ \frac{\Phi \vdash_{reg} \rho \preceq: \rho'}{\Phi \vdash \mathbf{Ref}_{\rho}\ \mathrm{T} \preceq: \mathbf{Ref}_{\rho'}\ \mathrm{T}}$$

$$\text{(TYPE-FUN)}\ \frac{\Phi \vdash \mathrm{T}'_0 \preceq: \mathrm{T}_0 \quad \Phi \vdash \mathrm{T}_1 \preceq: \mathrm{T}'_1 \quad \Phi \vdash_{eff} \sigma \preceq: \sigma'}{\Phi \vdash \mathrm{T}_0 \xrightarrow{\sigma} \mathrm{T}_1 \preceq: \mathrm{T}'_0 \xrightarrow{\sigma'} \mathrm{T}'_1}$$

Effect Subsumption: $\Phi \vdash_{eff} \sigma \preceq: \sigma'$

$$\text{(EFF-REFL)}\ \Phi \vdash_{eff} \sigma \preceq: \sigma \qquad \text{(EFF-TRAN)}\ \frac{\Phi \vdash_{eff} \sigma \preceq: \sigma_0 \quad \Phi \vdash_{eff} \sigma_0 \preceq: \sigma'}{\Phi \vdash_{eff} \sigma \preceq: \sigma'} \qquad \text{(EFF-SUB)}\ \frac{\sigma \subseteq \sigma'}{\Phi \vdash_{eff} \sigma \preceq: \sigma'} \qquad \text{(EFF-CONS)}\ \frac{\sigma \preceq: \sigma' \in \Phi}{\Phi \vdash_{eff} \sigma \preceq: \sigma'}$$

$$\text{(EFF-ACC)}\ \frac{\Phi \vdash_{reg} \rho \preceq: \rho'}{\Phi \vdash_{eff} \{acc_{\rho}\mathrm{T}\} \preceq: \{acc_{\rho'}\mathrm{T}\}} \qquad \text{(EFF–INST)}\ \frac{\forall \vec{g}.\Sigma \in \Phi \quad \sigma \preceq: \sigma' \in \theta\Sigma \text{ for some } \theta \quad dom(\theta) = \vec{g} \quad ran(\theta) \cap ftv(\Sigma) = \emptyset}{\Phi \vdash_{eff} \sigma \preceq: \sigma'}$$

Region Subsumption: $\Phi \vdash_{reg} \rho \preceq: \rho'$

$$\text{(REG-REFL)}\ \Phi \vdash_{reg} \rho \preceq: \rho \qquad \text{(REG-TRANS)}\ \frac{\Phi \vdash_{reg} \rho \preceq: \rho_0 \quad \Phi \vdash_{reg} \rho_0 \preceq: \rho'}{\Phi \vdash_{reg} \rho \preceq: \rho'} \qquad \text{(REG-SUB)}\ \frac{\rho \subseteq \rho'}{\Phi \vdash_{reg} \rho \preceq: \rho'} \qquad \text{(REG-CONS)}\ \frac{\rho \preceq: \rho' \in \Phi}{\Phi \vdash_{reg} \rho \preceq: \rho'}$$

$$\text{(REG–INST)}\ \frac{\forall \vec{g}.\Sigma \in \Phi \quad \rho \preceq: \rho' \in \theta\Sigma \text{ for some } \theta \quad dom(\theta) = \vec{g} \quad ran(\theta) \cap ftv(\Sigma) = \emptyset}{\Phi \vdash_{reg} \rho \preceq: \rho'}$$

Relationship Entailment: $\Phi \vdash_{ar} \Lambda$

$$\text{(REL–IN)}\ \frac{\Lambda \in \Phi}{\Phi \vdash_{ar} \Lambda} \qquad \text{(REL-CLOSED)}\ \frac{\Phi \vdash_{ar} \sigma\ \mathbb{R}\ \sigma' \quad \Phi \vdash_{eff} \sigma_0 \preceq: \sigma \quad \Phi \vdash_{eff} \sigma_1 \preceq: \sigma'}{\Phi \vdash_{ar} \sigma_0\ \mathbb{R}\ \sigma_1}$$

Figure 8 λ_{ie} Subsumption and Entailment.

designers should be able to find conceptual analogies in existing systems, such as privileges in Marino *et al.* [28].

Notations and Convenience Functions

We use (overloaded) function *ftv* to compute the set of free (*i.e.*, neither universally bound nor existentially bound) variables in T, ρ and σ. We use $fv(\mathrm{e})$ to compute the set of free variables in expression e. We use *dom* and *ran* to compute the domain and the range of a function. All definitions are standard. Substitution θ is a mapping function from type variables α to types T, region variables γ to regions ρ and effect variables ς to effects σ. The composition of substitutions, written $\theta\theta'$ is defined as $\theta\theta'(g) = \theta(\theta'(g))$.

We use comma for sequence concatenation. For example, $\Gamma, \mathrm{x} \mapsto \tau$ denotes appending sequence Γ with an additional binding from x to τ.

4.2 Subsumption and Entailment

Figure 8 defines subsumption relations for types, effects, and regions. All three forms of subsumption are reflexive and transitive. For function types, both return types and effects are covariant, whereas argument types are contra-variant. For **Ref** types, the regions are covariant, whereas the types for what the store holds must be invariant [37].

(EFF-INST) and (REG-INST) capture the instantiation of universal variables in subsumption context relationship. After all, the latter is a collection of "parameterized" subsumption relationships which can be instantiated.

Finally, we define a simple relation $\Phi \vdash_{ar} \Lambda$ to denote that relationship set Φ can entail Λ. (REL-IN) says any relationship set may entail its element. (REL-CLOSED) intuitively says that $\mathbb{R}$ is closed under taking subsetting. (REL-CLOSED) is a manifestation of the monotonic requirement [28, 8] for $\mathbb{R}$, introduced in Section 3.

4.3 Typing Judgment Overview

Typing judgment in our system takes the form of $\Phi; \Gamma \vdash \mathrm{e} : \mathrm{T}, \sigma$, which consists of a type environment Γ, a relationship set Φ, an expression e, its type T and effect σ. When the relationship set and the type environment are empty, we further shorten the judgment as $\vdash \mathrm{e} : \mathrm{T}, \sigma$ for convenience. The judgment is defined in Figure 9, with auxiliary definitions related to universal and existential quantification deferred to Figure 10.

The rules (T-LET) and (T-VAR) follow the familiar *let-polymorphism* (or Damas-Milner polymorphism [13]). Universal quantification is introduced at **let** boundaries, through the function $Gen(\Gamma, \sigma)(\mathrm{T})$. Its elimination is performed at (T-VAR), via $\preceq$. Both definitions are standard, and appear in Figure 10. The let-polymorphism in **let** $\mathrm{x} \ = \ \mathrm{e}$ **in** e' expression is sound because of the *Gen* function in the rule (T-LET). The *Gen* function enforces the standard value restriction [36]. That is, if e is a value, its type could be generalized and thus be polymorphic, otherwise its type will be monomorphic.

(T-SUB) describes subtyping, where both (monomorphic) type subsumption and effect subsumption may be applied. Rules (T-REF), (T-GET) and (T-SET) for store operations produce the effects of access rights **init**, **rd** and **wr**, respectively. All other rules other than (T-ASSUME) and (T-SAFE) carry little surprise for an effect system.

4.4 Static Typing for Dynamic Intensional Analysis

To demonstrate how intensional effect analysis works, let us first consider an unsound but intuitive notion of **assuming** typing in (T-ASSUME-UNSOUND) (below).

$$\text{(T-ASSUME-UNSOUND)} \quad \frac{\begin{array}{cc}\Phi; \Gamma \vdash \mathrm{e} : \mathrm{T}, \sigma & \Phi; \Gamma \vdash \mathrm{e}' : \mathrm{T}', \sigma' \\ \Phi, \sigma \ \mathbb{R} \ \sigma'; \Gamma \vdash \mathrm{e}_0 : \mathrm{T}'', \sigma_0 & \Phi; \Gamma \vdash \mathrm{e}_1 : \mathrm{T}'', \sigma_1\end{array}}{\Phi; \Gamma \vdash \mathbf{assuming} \ \mathrm{e} \ \mathbb{R} \ \mathrm{e}' \ \mathbf{do} \ \mathrm{e}_0 \ \mathbf{else} \ \mathrm{e}_1 : \mathrm{T}'', \sigma_0 \cup \sigma_1}$$

To type check the **do** expression e_0, (T-ASSUME-UNSOUND) takes advantage of the fact that expressions e and e' satisfy the relation $\mathbb{R}$, *i.e.*, in the third condition of the rule, we strengthen the current Φ with $\sigma \ \mathbb{R} \ \sigma'$. The (T-SAFE) rule in Figure 9 says that the expression type checks iff Φ entails the abstract effect relationship $\mathbb{R}$. As a result, a $\mathbb{SAFE}$ expression whose safety happens to rely on $\sigma \ \mathbb{R} \ \sigma'$ can be verified to be safe by the static system.

Albeit tempting, the rule above is unsound. To illustrate, consider the safe parallelism discipline in the following example, *i.e.*, we instantiate the $\mathbb{R}$ relation with noninterference relation # and the $\mathbb{SAFE}$ expression with parallel expression $||$.

Typing: $\Phi;\Gamma \vdash e : T, \sigma$

$$\text{(T-Bool)}\quad \Phi;\Gamma \vdash b : \mathbf{Bool}, \emptyset \qquad \text{(T-var)}\quad \frac{T \preceq \Gamma(x)}{\Phi;\Gamma \vdash x : T, \emptyset} \qquad \text{(T-let)}\quad \frac{\Phi;\Gamma \vdash e : T,\sigma \qquad \Phi;\Gamma, x \mapsto \mathit{Gen}(\Gamma,\sigma)(T) \vdash e' : T', \sigma'}{\Phi;\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : T', \sigma \cup \sigma'}$$

$$\text{(T-sub)}\quad \frac{\Phi;\Gamma \vdash e : T,\sigma \qquad \Phi \vdash T \preceq: T' \qquad \Phi \vdash_{\mathit{eff}} \sigma \preceq: \sigma'}{\Phi;\Gamma \vdash e : T', \sigma'} \qquad \text{(T-abs)}\quad \frac{\emptyset;\Gamma, x \mapsto T \vdash e : T', \sigma}{\Phi;\Gamma \vdash \lambda x : T.e : T \xrightarrow{\sigma} T', \emptyset}$$

$$\text{(T-app)}\quad \frac{\Phi;\Gamma \vdash e : T \xrightarrow{\sigma} T', \sigma' \qquad \Phi;\Gamma \vdash e' : T, \sigma''}{\Phi;\Gamma \vdash e\ e' : T', \sigma \cup \sigma' \cup \sigma''} \qquad \text{(T-ref)}\quad \frac{\Phi;\Gamma \vdash e : T,\sigma}{\Phi;\Gamma \vdash \mathbf{ref}\ \rho\ T\ e : \mathbf{Ref}_\rho\ T, \sigma \cup \mathbf{init}_\rho T}$$

$$\text{(T-get)}\quad \frac{\Phi;\Gamma \vdash e : \mathbf{Ref}_\rho\ T, \sigma}{\Phi;\Gamma \vdash !\, e : T, \sigma \cup \mathbf{rd}_\rho T} \qquad \text{(T-set)}\quad \frac{\Phi;\Gamma \vdash e : \mathbf{Ref}_\rho\ T, \sigma \qquad \Phi;\Gamma \vdash e' : T, \sigma'}{\Phi;\Gamma \vdash e := e' : T, \sigma \cup \sigma' \cup \mathbf{wr}_\rho T}$$

$$\text{(T-if-then-else)}\quad \frac{\Phi;\Gamma \vdash e : \mathbf{Bool}, \sigma \qquad \Phi;\Gamma \vdash e_0 : T, \sigma_0 \qquad \Phi;\Gamma \vdash e_1 : T, \sigma_1}{\Phi;\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ e_0\ \mathbf{else}\ e_1 : T, \sigma \cup \sigma_0 \cup \sigma_1}$$

$$\text{(T-assume)}\quad \frac{\begin{array}{c} \overline{x} = \mathit{fv}(e) \cup \mathit{fv}(e') \qquad \Gamma(\overline{x}) = \overline{\tau} \qquad \Phi'' \vdash \mathit{EGen}(\overline{\tau}) \Rightarrow \overline{\tau}' \qquad \Gamma' = \Gamma, \overline{x \mapsto \tau'} \qquad \Phi' = \Phi, \Phi'' \\ \Phi';\Gamma' \vdash e : T, \sigma \qquad \Phi';\Gamma' \vdash e' : T', \sigma' \\ \Phi', \sigma\ \mathbb{R}\ \sigma';\Gamma' \vdash e_0 : T''', \sigma_2 \qquad \Phi \vdash T''' \uparrow T'' \qquad \Phi \vdash \sigma_2 \uparrow \sigma_0 \qquad \Phi;\Gamma \vdash e_1 : T'', \sigma_1 \end{array}}{\Phi;\Gamma \vdash \mathbf{assuming}\ e\ \mathbb{R}\ e'\ \mathbf{do}\ e_0\ \mathbf{else}\ e_1 : T'', \sigma_0 \cup \sigma_1}$$

$$\text{(T-safe)}\quad \frac{\Phi;\Gamma \vdash e : T_0, \sigma_0 \qquad \Phi;\Gamma \vdash e' : T_1, \sigma_1 \qquad \Phi \vdash_{\overline{ar}} \sigma_0\ \mathbb{R}\ \sigma_1 \qquad \mathit{clientT}(T, \sigma, T_0, \sigma_0, T_1, \sigma_1)}{\Phi;\Gamma \vdash \mathbb{SAFE}\ e\ e' : T, \sigma}$$

Figure 9 λ_{ie} Typing Rules.

▶ **Example 4** (Soundness Challenge)**.** In the following example, the variables `x` and `y` have the same static (but different dynamic) type. Thus, the expression `x 3` and `y 3` have the same static effect. Should the parallel expression at line 5 typecheck with the assumption expression at line 4, there would be a data race at runtime.

```
1 let buff = ref 0 in
2  let x = if 1 > 0 then λz. !buff else λz. buff := z in
3   let y = if 0 > 1 then λz. !buff else λz. buff := z in
4    assuming !buff  # x 3
5          do !buff  || y 3
6        else !buff  ; x 2
```

In this example, we have an imperative reference `buff`, and two structurally similar but distinct functions `x` and `y`. The code intends to perform parallelization, *i.e.*, `buff || y 3`, line 5. Let us review the types of the variables:

$$\begin{array}{rcl} \texttt{buff} & : & \mathbf{Ref}_\rho\ \mathbf{Int} \\ \texttt{x} & : & \mathbf{Int} \xrightarrow{\{\mathbf{rd}_\rho \mathbf{Int}, \mathbf{wr}_\rho \mathbf{Int}\}} \mathbf{Int} \\ \texttt{y} & : & \mathbf{Int} \xrightarrow{\{\mathbf{rd}_\rho \mathbf{Int}, \mathbf{wr}_\rho \mathbf{Int}\}} \mathbf{Int} \end{array}$$

∀ Introduction: **Gen**	$Gen(\Gamma,\sigma)(\mathrm{T}) = \forall\vec{g}.\mathrm{T}$ where $\vec{g} = ftv(\mathrm{T})\backslash(ftv(\Gamma)\cup ftv(\sigma))$
∀ Elimination: $\preceq$	$\mathrm{T}' \preceq \forall\vec{g}.\mathrm{T}$ if $\mathrm{T}' = \theta\mathrm{T}$ for some θ
∃ Introduction: **EGen**	

$$\begin{array}{lcll} \mathbb{P} & ::= & - \mid \mathbf{Ref}_{\mathbb{PR}}\ \mathrm{T} \mid \mathrm{T} \xrightarrow{\mathbb{PE}} \mathrm{T} \mid \mathrm{T} \xrightarrow{\sigma} \mathbb{P} & \textit{pack context} \\ \mathbb{PE} & ::= & - \mid \overline{\varpi}, \mathbb{PE}, \overline{\varpi}' \mid acc_{\mathbb{PR}}\ \mathrm{T} \mid acc_\rho\ \mathbb{P} & \\ \mathbb{PR} & ::= & - \mid \overline{\zeta}, \mathbb{PR}, \overline{\zeta}' & \end{array}$$

$$\begin{array}{rcll} EGen(\forall\vec{g}.\mathrm{T}) & \triangleq & \forall\vec{g}.EGenM(\mathrm{T},\emptyset) & \\ EGenM(\mathbb{PE}[\sigma],\vec{g}) & \triangleq & \exists\varsigma \preceq: \sigma.EGenM(\mathbb{PE}[\varsigma \preceq: \sigma], \vec{g}\cup\{\varsigma\}) & \text{if } \sigma\notin\vec{g}, ftv(\sigma)\subseteq\vec{g}, \varsigma \text{ fresh} \\ EGenM(\mathbb{PR}[\rho],\vec{g}) & \triangleq & \exists\gamma \preceq: \rho.EGenM(\mathbb{PR}[\gamma \preceq: \rho], \vec{g}\cup\{\gamma\}) & \text{if } \rho\notin\vec{g}, ftv(\rho)\subseteq\vec{g}, \gamma \text{ fresh} \\ EGenM(\mathrm{T},\vec{g}) & \triangleq & \mathrm{T} & \text{if } \sigma\in\vec{g} \text{ for any } \mathrm{T} = \mathbb{PE}[\sigma] \\ & & & \quad \rho\in\vec{g} \text{ for any } \mathrm{T} = \mathbb{PR}[\rho] \end{array}$$

∃ Elimination: ⇒	$\forall\vec{g}.(\theta\Sigma) \vdash \forall\vec{g}.\exists\Sigma.\mathrm{T} \Rightarrow \forall\vec{g}.\theta\mathrm{T}$ for some $\theta \wedge dom(\theta)\subseteq\vec{g}$
Lifting: ↑	

$$\begin{array}{r} \Phi\vdash\mathbb{P}[\varsigma \preceq: \sigma]\uparrow\mathbb{P}[\sigma'] \text{ if } \forall\vec{g}.\Sigma\in\Phi, \varsigma \preceq: \sigma\in\theta\Sigma \text{ for some } \theta, \Phi\vdash\sigma\uparrow\sigma' \\ \Phi\vdash\mathbb{P}[\gamma \preceq: \rho]\uparrow\mathbb{P}[\rho'] \text{ if } \forall\vec{g}.\Sigma\in\Phi, \gamma \preceq: \rho\in\theta\Sigma \text{ for some } \theta, \Phi\vdash\rho\uparrow\rho' \\ \Phi\vdash\mathrm{T}\uparrow\mathrm{T} \text{ if } \forall\vec{g}.\Sigma\in\Phi, ftv(\mathrm{T})\cap(\forall\vec{g}.\Sigma)=\emptyset \end{array}$$

$$\begin{array}{r} \Phi\vdash\mathbb{PE}[\varsigma \preceq: \sigma]\uparrow\mathbb{PE}[\sigma'] \text{ if } \forall\vec{g}.\Sigma\in\Phi, \varsigma \preceq: \sigma\in\theta\Sigma \text{ for some } \theta, \Phi\vdash\sigma\uparrow\sigma' \\ \Phi\vdash\mathbb{PE}[\gamma \preceq: \rho]\uparrow\mathbb{PE}[\rho'] \text{ if } \forall\vec{g}.\Sigma\in\Phi, \gamma \preceq: \rho\in\theta\Sigma \text{ for some } \theta, \Phi\vdash\rho\uparrow\rho' \\ \Phi\vdash\sigma\uparrow\sigma \text{ if } \forall\vec{g}.\Sigma\in\Phi, ftv(\sigma)\cap(\forall\vec{g}.\Sigma)=\emptyset \end{array}$$

$$\begin{array}{r} \Phi\vdash\mathbb{PR}[\gamma \preceq: \rho]\uparrow\mathbb{PR}[\rho'] \text{ if } \forall\vec{g}.\Sigma\in\Phi, \gamma \preceq: \rho\in\theta\Sigma \text{ for some } \theta, \Phi\vdash\rho\uparrow\rho' \\ \Phi\vdash\rho\uparrow\rho \text{ if } \forall\vec{g}.\Sigma\in\Phi, ftv(\rho)\cap(\forall\vec{g}.\Sigma)=\emptyset \end{array}$$

Figure 10 Definitions for ∀ and ∃ Introduction and Elimination.

and the effects of the expressions:

$$\begin{array}{rcl} \texttt{!buff} & : & \{\mathbf{rd}_\rho\mathbf{Int}\} \\ \texttt{x 3} & : & \{\mathbf{rd}_\rho\mathbf{Int}, \mathbf{wr}_\rho\mathbf{Int}\} \\ \texttt{y 3} & : & \{\mathbf{rd}_\rho\mathbf{Int}, \mathbf{wr}_\rho\mathbf{Int}\} \end{array}$$

According to the static system, the types of `x` and `y` are exactly the same. Thus, by the third condition of (T-SAFE), the expression `buff || y 3` on line 5, is well-formed. This is because the **assuming** expression has placed $\{\mathbf{rd}_\rho\mathbf{Int}\}$ # $\{\mathbf{rd}_\rho\mathbf{Int}, \mathbf{wr}_\rho\mathbf{Int}\}$ as an element of the relationship set, after typechecking `buff # x 3` on line 4.

At runtime, the initialization expression of the **let** expressions will be first evaluated before being assigned to the variables (*call-by-value*, details in Section 5). Therefore, `x` becomes λ`z.!buff` and `y` becomes λ`z.buff:=z` before the **assuming** expression. Informally, we refer to the effect computed at runtime through dynamic typing (*e.g.*, right before the **assuming** expression) as *dynamic effect*, as opposed to the *static effect* computed at compile time. The dynamic types of the variables are:

$$\begin{array}{lcl} \texttt{x} & : & \mathbf{Int} \xrightarrow{\{\mathbf{rd}_\rho\mathbf{Int}\}} \mathbf{Int} \\ \texttt{y} & : & \mathbf{Int} \xrightarrow{\{\mathbf{wr}_\rho\mathbf{Int}\}} \mathbf{Int} \end{array}$$

and effects of the relevant expressions are:

$$\begin{array}{rcl} \texttt{!buff} & : & \{\mathbf{rd}_\rho\mathbf{Int}\} \\ \texttt{x 3} & : & \{\mathbf{rd}_\rho\mathbf{Int}\} \\ \texttt{y 3} & : & \{\mathbf{wr}_\rho\mathbf{Int}\} \end{array}$$

Clearly, the dynamic effect computed for `buff` and that for `x 3` on line 4 do not conflict. Therefore, the **do** expression `buff || y 3` will be evaluated. However, the effects of the expressions `buff` and `y 3` on line 5 do conflict, which causes unsafe parallelism.

The root cause of the problem is that the static and the dynamic system make decisions based on two related but different effects: one with the static effect, and the other with the dynamic effect. A sound type system must be able to differentiate the two.

A Sound Design with Bounded Existentials

The key insight from the discussion above is that the static system must be able to express the *dynamic effect* that the **assuming** expression makes decision upon. Before we move on, let us first state several simple observations:

(i) (Dynamic effect refines static effect) The static effect of an expression e is a conservative approximation of the dynamic effect.

(ii) (Free variables determine effect difference) Improved precision of intensional effect polymorphism is achieved by using the more precise types for the free variables, see *e.g.*, dynamic and static type of the variable `x` in Example 4.

Observation *(i)* indicates the possibility of referring to the dynamic effect as "there exists some effect that is subsumed by the static effect." Observation *(ii)* further suggests that dynamic effect can be computed by treating all free variables existentially: "there exists some type T which is a subtype of the static type T′ for each free variable, to help mimic the type environment while dynamic effect is computed". Bounded existential types provide an ideal vehicle for expressing this intention.

(T-Assume) captures the type checking of an **assuming** expression. We substitute the type of each free variable with (an instance of) its existential counterpart. Let us revisit Example 4, this time with (T-Assume). The free variables of the **assuming** expression at line 4, in Example 4 are `x` and `buff`. The original types of the free variables are:

$$\texttt{buff} : \mathbf{Ref}_\rho\ \mathbf{Int} \qquad \text{and} \qquad \texttt{x} : \mathbf{Int} \xrightarrow{\{\mathbf{rd}_\rho\mathbf{Int},\mathbf{wr}_\rho\mathbf{Int}\}} \mathbf{Int}$$

The existential types used to type check the **assuming** expressions are:

$$\texttt{buff} : \mathbf{Ref}_\rho\ \mathbf{Int} \quad \text{and} \quad \texttt{x} : \exists\varsigma_1 \preceq: \{\mathbf{rd}_\rho\mathbf{Int}\}, \varsigma_2 \preceq: \{\mathbf{wr}_\rho\mathbf{Int}\}.\mathbf{Int} \xrightarrow{\varsigma_1,\varsigma_2} \mathbf{Int}$$

The relationship set is:

$$\Phi = \varsigma_1 \preceq: \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \preceq: \mathbf{wr}_\rho\mathbf{Int} \tag{1}$$

The effects of the condition expressions are:

$$\texttt{!buff} : \mathbf{rd}_\rho\mathbf{Int} \qquad \text{and} \qquad \texttt{x 3} : \varsigma_1 \preceq: \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \preceq: \mathbf{wr}_\rho\mathbf{Int}$$

Parallelism: $\|\|$	$clientT(\mathrm{T}, \sigma, \mathrm{T}_0, \sigma_0, \mathrm{T}_1, \sigma_1)$	$\stackrel{\text{def}}{=}$	$(\mathrm{T} = \mathrm{T}_0 = \mathrm{T}_1) \wedge (\sigma = \sigma_0 \cup \sigma_1)$
Security: **exec**	$clientT(\mathrm{T}, \sigma, \mathrm{T}_0, \sigma_0, \mathrm{T}_1, \sigma_1)$	$\stackrel{\text{def}}{=}$	$(\mathrm{T} = \mathrm{T}_1) \wedge (\sigma = \sigma_1)$
UI: **spawn**	$clientT(\mathrm{T}, \sigma, \mathrm{T}_0, \sigma_0, \mathrm{T}_1, \sigma_1)$	$\stackrel{\text{def}}{=}$	$(\mathrm{T} = \mathbf{void}) \wedge (\sigma = \emptyset)$
Memoization: **lookup**	$clientT(\mathrm{T}, \sigma, \mathrm{T}_0, \sigma_0, \mathrm{T}_1, \sigma_1)$	$\stackrel{\text{def}}{=}$	$(\mathrm{T} = \mathrm{T}_1) \wedge (\sigma = \sigma_0)$

Figure 11 Client Implementation of Predicate *clientT*.

To type check the **do** expression, Φ is strengthened as:

$$\Phi' = \{\mathbf{rd}_\rho\mathbf{Int} \;\#\; \{\varsigma_1 \preceq: \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \preceq: \mathbf{wr}_\rho\mathbf{Int}\}, \varsigma_1 \preceq: \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \preceq: \mathbf{wr}_\rho\mathbf{Int}\} \quad (2)$$

When type checking the expression on line 5, `y 3` has effect $\{\mathbf{rd}_\rho\mathbf{Int}, \mathbf{wr}_\rho\mathbf{Int}\}$. We cannot establish $\vdash_{ar}$ (as Figure 8). A type error is correctly induced against the potential unsafe parallel expression.

Rule (T-ASSUME) first computes the free variables from the two condition expressions, written $\overline{\mathrm{x}} = fv(\mathrm{e}) \cup fv(\mathrm{e}')$. With assumption $\Gamma(\overline{\mathrm{x}}) = \overline{\tau}$, all free variables $\overline{\mathrm{x}}$ are considered for type environment strengthening. It then applies the existential *introduction* function *EGen* to strengthen τ', the bounded existential with the original type τ as the bound. The definition of *EGen* is in Figure 10. It then *eliminates* (or *open*) the existential quantification using $\Rightarrow$. In a nutshell, this predicate $\Phi'' \vdash EGen(\overline{\tau}) \Rightarrow \overline{\tau}'$ introduces an existential type and eliminates it right away (a common strategy in building abstract data types [30]). Subsumption relationship information is placed into the relationship set, $\Phi' = \Phi, \Phi''$. The new environment Γ' has the new types $\overline{\tau}'$ for the free variables, an instantiation of the bounded existential type.

Function *EGen* uses the *EGenM* function to quantify effects and regions. Here, to produce the existential type, function *EGen* maintains the structure of the original type, *e.g.*, if the original type is a function type, it produces a new function type with all *covariant* types/effects/regions quantified. Observe that *contravariant* types/effects/regions are harmless: their dynamic counterpart (which also refines the static one) does not cause soundness problems. To facilitate the quantification (also known as existential introduction or *packed* [30]), three *pack contexts*, $\mathbb{P}$, $\mathbb{PE}$, $\mathbb{PR}$, are defined, representing the contexts to contain a type, an effect, or a region, respectively.

Finally, the type of the **do** expression needs to be *lifted*, weakening types that may potentially contain refreshed generic variables of existential types, through a self-explaining $\uparrow$ definition in Figure 10. For example, the effect computed for the expression `x 3` is $\varsigma_1 \preceq: \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \preceq: \mathbf{wr}_\rho\mathbf{Int}$. The $\uparrow$ function applies the substitution of $\{\varsigma_1 \mapsto \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \mapsto \mathbf{wr}_\rho\mathbf{Int}\}$ on the precomputed effect and produces static effect $\mathbf{rd}_\rho\mathbf{Int}, \mathbf{wr}_\rho\mathbf{Int}$.

The typing of (T-SAFE) relies on the client function *clientT*. $clientT(\mathrm{T}, \sigma, \mathrm{T}_0, \sigma_0, \mathrm{T}_1, \sigma_1)$ defines the conditions where a safe expression should typecheck, as shown in Figure 11.

5 Dynamic Semantics

This section describes the dynamic semantics of λ_{ie}. The highlight is to support a highly precise notion of effect polymorphism via a *lightweight* notion of dynamic typing, which we call *differential alignment*.

Definitions:

s	$::=$	$\overline{l \rightarrow_{\langle \rho, \mathrm{T} \rangle} v}$	*store*
f	$::=$	$\overline{acc(l)}$	*trace*
v	$::=$	$\ldots \mid l$	*(extended) values*
$\mathbb{E}$	$::=$	$- \mid \mathbb{E}\ \mathrm{e} \mid v\ \mathbb{E} \mid \textbf{let}\ \mathrm{x} = \mathbb{E}\ \textbf{in}\ \mathrm{e} \mid \textbf{let}\ \mathrm{x} = v\ \textbf{in}\ \mathbb{E} \mid \textbf{ref}\ \rho\ \mathrm{T}\ \mathbb{E}$ $\mid !\mathbb{E} \mid \mathbb{E} := \mathrm{e} \mid v := \mathbb{E} \mid \textbf{if}\ \mathbb{E}\ \textbf{then}\ \mathrm{e}\ \textbf{else}\ \mathrm{e}$	*evaluation context*

Dynamic Typing: $s; \Phi; \Gamma \vdash_{\mathrm{D}} \mathrm{e} : \mathrm{T}, \sigma$

$$\text{(DT-LOC)}\quad \frac{\{l \mapsto_{\langle \rho, \mathrm{T} \rangle} v\} \in s}{s; \Phi; \Gamma \vdash_{\mathrm{D}} l : \mathbf{Ref}_{\rho}\ \mathrm{T}, \emptyset}$$

For all other (DT-*) rules, each is isomorphic to its counterpart (T-*) rule, except that every occurrence of judgment $\Phi; \Gamma \vdash \mathrm{e} : \mathrm{T}, \sigma$ in the latter rule should be substituted with $s; \Phi; \Gamma \vdash_{\mathrm{D}} \mathrm{e} : \mathrm{T}, \sigma$ in the former.

Evaluation relation: $s; \mathrm{e}; f \rightarrow s'; \mathrm{e}'; f'$

(*cxt*)	$s; \mathbb{E}[\mathrm{e}]; f \rightarrow s'; \mathbb{E}[\mathrm{e}']; f, f'$	if $s; \mathrm{e} \Rightarrow s'; \mathrm{e}'; f'$
(*asm*)	$s; \textbf{assuming}\ \mathrm{e}_1\ \mathbb{R}\ \mathrm{e}_2\ \textbf{do}\ \mathrm{e}\ \textbf{else}\ \mathrm{e}' \Rightarrow s; \mathrm{e}_0; \emptyset$	if $s; \emptyset; \emptyset \vdash_{\mathrm{D}} \mathrm{e}_i : \mathrm{T}_i, \sigma_i$ for $i = 1, 2$ and $\mathrm{e}_0 = \begin{cases} \mathrm{e} & \text{if } \sigma_1\ \mathbb{R}\ \sigma_2 \\ \mathrm{e}' & \text{otherwise} \end{cases}$
(*safe*)	$s; \mathbb{SAFE}\ \mathrm{e}\ \mathrm{e}' \Rightarrow clientR(s, \mathrm{e}, \mathrm{e}')$	
(*set*)	$s; l := v \Rightarrow s, \{l \mapsto_{\langle \rho, \mathrm{T} \rangle} v\}; v; \mathbf{wr}(l)$	if $\{l \mapsto_{\langle \rho, T \rangle} v'\} \in s$
(*ref*)	$s; \textbf{ref}\ \rho\ \mathrm{T}\ v \Rightarrow s, \{l \mapsto_{\langle \rho, \mathrm{T} \rangle} v\}; l; \mathbf{init}(l)$	if l *fresh*
(*get*)	$s; !l \Rightarrow s; s(l); \mathbf{rd}(l)$	
(*app*)	$s; \lambda \mathrm{x} : \mathrm{T}.\mathrm{e}\ v \Rightarrow s; [\mathrm{x} \mapsto v]\mathrm{e}; \emptyset$	
(*let*)	$s; \textbf{let}\ \mathrm{x} = v\ \textbf{in}\ \mathrm{e} \Rightarrow s; [\mathrm{x} \mapsto v]\mathrm{e}; \emptyset$	
(*ifT*)	$s; \textbf{if}\ true\ \textbf{then}\ \mathrm{e}\ \textbf{else}\ \mathrm{e}' \Rightarrow s; \mathrm{e}; \emptyset$	
(*ifF*)	$s; \textbf{if}\ false\ \textbf{then}\ \mathrm{e}\ \textbf{else}\ \mathrm{e}' \Rightarrow s; \mathrm{e}'; \emptyset$	

Figure 12 λ_{ie} Operational Semantics.

Operational Semantics Overview

The λ_{ie} runtime configuration consists of a *store* s, the to-be-evaluated expression e, and a *trace* f, defined in Figure 12. The store maps references (or locations) l to values v. In addition to booleans and functions, locations themselves are values as well. Each store cell also records the region ρ and type T information of the reference. A *trace* can informally be viewed as "realized effects," and it is defined as a sequence of accesses to references, with $\mathbf{init}(l)$, $\mathbf{rd}(l)$, and $\mathbf{wr}(l)$, denoting the instantiation, read, and write to location l respectively. Traces are only needed to demonstrate the properties of our language. This structure and its runtime maintenance are unnecessary in a λ_{ie} implementation.

The small-step semantics is defined by relation $s; \mathrm{e}; f \rightarrow s'; \mathrm{e}'; f'$, which says that the evaluation of an expression e with the store s and trace f results in the new expression e', the new store s', and the new trace f'. We use notation $[\mathrm{x} \mapsto v]\mathrm{e}$ to define the substitution of x with v of expression e. We use $\rightarrow^*$ to represent the reflexive and transitive closure of $\rightarrow$.

Dynamic Effect Inspection

Most reduction rules are conventional, except *(asm)* and *(safe)*. The *(asm)* rule captures the essence of the **assuming** expression, which relies on dynamic typing to achieve dynamic effect inspection. Dynamic typing is defined through type derivation $s; \Phi; \Gamma \vdash_{D} \mathrm{e} : \mathrm{T}, \sigma$, defined in the same figure, which extends static typing with one additional rule for reference value typing.

At runtime, the **assuming** expression retrieves the more precise dynamic effect of expression e_1 and e_2, and checks whether relation $\mathbb{R}$ holds. Observe that at runtime, e_1 and e_2 in the **assuming** expression are not identical to their respective forms when the program is written. Now, the free variables in the static program have been substituted with values, which carry more precise information on types, regions, and effects. This is the root cause why intensional effect polymorphism can achieve higher precision than a purely static effect system.

It should be noted that we evaluate neither e_1 nor e_2 at the evaluation of the **assuming** expression. In other words, λ_{ie} is not an *a posteriori* effect monitoring system.

The reduction of *(safe)* relies on an abstract function *clientR*. $\mathit{clientR}(s, \mathrm{e}, \mathrm{e}')$ computes the runtime configuration after the one-step evaluation of the $\mathbb{SAFE}$ expression. For example, for the information security example (Figure 2), the configuration $s; \mathit{adv.show}(\mathit{this}); \emptyset$ will be the result of the *clientR* function. The abstract treatment of this function allows λ_{ie} to be defined in a highly modular fashion, similar to previous work [28]. We will come back to this topic, especially its impact on soundness, in Sec. 6.

Optimization: Efficient Effect Inspection via Differential Alignment

The reduction system we have introduced so far may not be efficient: it requires full-fledged dynamic typing, which may entail dynamic construction of type derivations to compute the dynamic effects. In this section, we introduce one optimization.

As observed in Section 4.4, the (sub)expressions that do not have free variables will have the same static effects (*i.e.*, computed via static typing) and dynamic effects (*i.e.*, computed via dynamic typing). Our key insight is that, the only "difference" between the two forms of effects for the same expression lies with those introduced by free variables in the expression. As a result, we define a new dynamic effect computation strategy with two steps:

1. At compile time, we compute the static effects of the two expressions used for the effect inspection of each **assuming** expression in the program. In the meantime, we record the type (which contains free type/effect/region variables) of each free variable that appears in these two expressions.
2. At runtime, we "align" the static type of each free variable with the dynamic type associated with the corresponding value that substitutes for that free variable. The alignment will compute a substitution of (static) type/effect/region variables to their dynamic counterparts. The substitution will then be used to substitute the effect we computed in Step 1 to produce the dynamic effect.

For Step 1, we define a transformation from expression e to an *annotated expression* d, defined in Figure 13. The two forms are identical, except that the **assuming** expression in the "annotated expression" now takes the form of **assuming** $(\overrightarrow{\mathrm{x} : \tau})\ \mathrm{e}_1 : \sigma_1\ \mathbb{R}\ \mathrm{e}_2 : \sigma_2$ **do** e **else** e', which records the free variables of expressions e_1 and e_2 and their corresponding static types, denoted as $\overrightarrow{\mathrm{x} : \tau}$. The same expression also records the statically computed effects σ_1 and σ_2 for e_1 and e_2. The free variable computation function *fv* and variable substitution

Abstract Syntax in Optimized λ_{ie}

$$
\begin{array}{lll}
\mathrm{d} ::= & v \mid \mathrm{x} \mid \mathrm{d}\ \mathrm{d} \mid \mathbf{let}\ \mathrm{x} = \mathrm{d}\ \mathbf{in}\ \mathrm{d} \mid \mathbf{if}\ \mathrm{d}\ \mathbf{then}\ \mathrm{d}\ \mathbf{else}\ \mathrm{d} & \textit{annotated expressions} \\
& \mid \mathbf{ref}\ \rho\ \mathrm{T}\ \mathrm{d} \mid !\mathrm{d} \mid \mathrm{d} := \mathrm{d} & \\
& \mid \mathbf{assuming}\ (\overrightarrow{\mathrm{x} : \tau})\ \mathrm{d} : \sigma\ \mathbb{R}\ \mathrm{d} : \sigma\ \mathbf{do}\ \mathrm{d}\ \mathbf{else}\ \mathrm{d} \mid \mathbb{SAFE}\ \mathrm{d}\ \mathrm{d} &
\end{array}
$$

Transformation: $\mathrm{e} \overset{\Phi,\Gamma}{\rightsquigarrow} \mathrm{d}$

$$
\begin{array}{rcll}
\mathrm{x} & \overset{\Phi,\Gamma}{\rightsquigarrow} & \mathrm{x} & \\
\mathrm{e}\ \mathrm{e}' & \overset{\Phi,\Gamma}{\rightsquigarrow} & \mathrm{d}\ \mathrm{d}' & \text{if } \mathrm{e} \overset{\Phi,\Gamma}{\rightsquigarrow} \mathrm{d}, \mathrm{e}' \overset{\Phi,\Gamma}{\rightsquigarrow} \mathrm{d}' \\
& \vdots & & \\
\mathbf{assuming}\ \mathrm{e}_1\ \mathbb{R}\ \mathrm{e}_2 & \overset{\Phi,\Gamma}{\rightsquigarrow} & \mathbf{assuming}\ (\overrightarrow{\mathrm{x} : \tau})\ \mathrm{d}_1 : \sigma_1\ \mathbb{R}\ \mathrm{d}_2 : \sigma_2 & \text{if } \overline{\mathrm{x}} = \mathit{fv}(\mathrm{e}_1) \cup \mathit{fv}(\mathrm{e}_2),\quad \Gamma(\overline{\mathrm{x}}) = \overline{\tau}' \\
\mathbf{do}\ \mathrm{e}_3\ \mathbf{else}\ \mathrm{e}_4 & & \mathbf{do}\ \mathrm{d}_3\ \mathbf{else}\ \mathrm{d}_4 & \Phi'' \vdash \mathit{EGen}(\overline{\tau}') \Rightarrow \overline{\tau}, \\
& & & \Phi' = \Phi, \Phi'' \\
& & & \Phi'; \Gamma, \overline{\mathrm{x} \mapsto \tau} \vdash \mathrm{d}_i : \mathrm{T}_i, \sigma_i \text{ for } i = 1,2 \\
& & & \mathrm{e}_i \overset{\Phi,\Gamma}{\rightsquigarrow} \mathrm{d}_i \text{ for } i = 1,2,3,4
\end{array}
$$

Operational Semantics in Optimized λ_{ie}: $\quad s; \mathrm{d}; f \rightarrow_O s; \mathrm{d}; f$

$$
\begin{array}{llcll}
(\mathit{Ocxt}) & s; \mathbb{E}[\mathrm{d}]; f & \rightarrow_O & s'; \mathbb{E}[\mathrm{d}']; f, f' & \text{if } s; \mathrm{d}; f \Rightarrow_O s'; \mathrm{d}'; f' \\
(\mathit{Oasm}) & s; \mathbf{assuming}(\overrightarrow{v : \tau}) & \Rightarrow_O & s; \mathrm{d}_0; \emptyset & \text{if } s; \emptyset; \emptyset \vdash_{\mathrm{DO}} \overline{v : \mathrm{T}, \emptyset} \\
& \mathrm{d}_1 : \sigma_1\ \mathbb{R}\ \mathrm{d}_2 : \sigma_2 & & & \text{and } \theta\overline{\tau} = \mathit{Gen}(\emptyset, \emptyset)(\overline{\mathrm{T}}) \\
& \mathbf{do}\ \mathrm{d}\ \mathbf{else}\ \mathrm{d}' & & & \text{and } \mathrm{d}_0 = \begin{cases} \mathrm{d} & \text{if } \theta\sigma_1\ \mathbb{R}\ \theta\sigma_2 \\ \mathrm{d}' & \text{otherwise} \end{cases}
\end{array}
$$

For all other $\Rightarrow_O$ rules, each is isomorphic to its counterpart $\Rightarrow$ rule, except that every occurrence of metavariable e in the latter rule should be substituted with d in the former.

Figure 13 Optimized λ_{ie} with Differential Alignment.

function are defined for d elements in an analogous fashion as for e elements. We omit these definitions.

Considering all the annotated information is readily available while we perform static typing of the **assuming** expression– as in (T-Assume) – the transformation from expression e to annotated expression d under Φ and Γ, denoted as $\mathrm{e} \overset{\Phi,\Gamma}{\rightsquigarrow} \mathrm{d}$, is rather predictable, defined in the same Figure.

The most interesting part of our optimized system is its dynamic semantics. Here we define a reduction system $\rightarrow_O$, at the bottom of the same figure. We further use $\rightarrow_O^*$ to represent the reflexive and transitive closure of $\rightarrow_O$. Upon the evaluation of the annotated **assuming** expression, the types associated with the free variables – now substituted with values – are "aligned" with the types associated with the corresponding values. The latter is computed by judgment $s; \Phi; \Gamma \vdash_{\mathrm{DO}} \mathrm{d} : \mathrm{T}, \sigma$, defined as $s; \Phi; \Gamma \vdash_{\mathrm{D}} \mathrm{e} : \mathrm{T}, \sigma$ where $\mathrm{e} \overset{\Phi,\Gamma}{\rightsquigarrow} \mathrm{d}$. In other words, we only need to dynamically type *values* in the optimized λ_{ie}. The alignment is achieved through the computation of the substitution θ. As we shall see in the next section, such a substitution always exists for well-typed programs.

6 Meta-Theories

In this section, we establish formal properties of λ_{ie}. We first show our type system is sound relative to sound customizations of the client effect systems (Section 6.1). We next present important soundness results for intensional effect polymorphism in Section 6.2, and a soundness and completeness result on differential alignment in Section 6.3. The proofs of these theorems and lemmas can be found in the accompanying technical report [25]. Before we proceed, let us first define two simple definitions that will be used for the rest of the section.

▶ **Definition 5.** [Redex Configuration] We say $< s; \mathrm{e}; f >$ is a redex configuration of program e', written $\mathrm{e}' \trianglerighteq \lessdot s, \mathrm{e}, f \gtrdot$, iff $\emptyset; \mathrm{e}'; \emptyset \rightarrow^* s; \mathbb{E}[\mathrm{e}]; f$.

Next, let us define relation $s \vdash f : \sigma$, which says that dynamic trace f *realizes* static effect σ under store s:

▶ **Definition 6.** [Effect-Trace Consistency] $s \vdash f : \sigma$ holds iff $acc(l) \in f$ implies $acc_\rho \mathrm{T} \in \sigma$ where $\{l \mapsto_{\langle \rho, \mathrm{T} \rangle} v\} \in s$.

6.1 Type Soundness

Our type system leaves the definition of $\mathbb{R}$ and $\mathbb{SAFE}$ e e$'$ abstract, both in terms of syntax and semantics. As a result, the soundness of our type system is conditioned upon how these definitions are concretized. Now let us explicitly define the sound concretization condition:

▶ **Definition 7** (Sound Client Concretization)**.** We say a λ_{ie} client is sound if under that concretization, the following condition holds: if $s; \Phi; \Gamma \vdash_{\mathrm{D}} \mathrm{e}_0 : \mathrm{T}_0, \sigma_0$, $s; \Phi; \Gamma \vdash_{\mathrm{D}} \mathrm{e}_1 : \mathrm{T}_1, \sigma_1$, $clientT(\mathrm{T}, \sigma,$
$\mathrm{T}_0, \sigma_0, \mathrm{T}_1, \sigma_1)$ and $(s', \mathrm{e}, f) = clientR(s, \mathrm{e}_0, \mathrm{e}_1)$, then $\mathrm{s}'; \Phi; \Gamma \vdash_{\mathrm{D}} \mathrm{e} : \mathrm{T}, \sigma$ and $s' \vdash f : \sigma$.

All lemmas and theorems for the rest of this section are implicitly under the assumption that Definition 7 holds, which we do not repeatedly state.

Our soundness proof is constructed through subject reduction and progress:

▶ **Lemma 8** (Type Preservation)**.** *If* $s; \Phi; \Gamma \vdash_D e : T, \sigma$ *and* $s; e; f \rightarrow s'; e'; f'$, *then* $s'; \Phi; \Gamma \vdash_D e' : T', \sigma'$ *and* $T' \preceq: T$ *and* $\sigma' \subseteq \sigma$.

▶ **Lemma 9** (Progress)**.** *If* $s; \Phi; \Gamma \vdash_D e : T, \sigma$ *then either* e *is a value, or* $s; e; f \rightarrow s'; e'; f'$ *for some* s', e', f'.

▶ **Theorem 10** (Type Soundness)**.** *Given an expression* e, *if* $\emptyset; \emptyset \vdash e : T, \sigma$, *then either the evaluation of* e *diverges, or there exist some* s, v, *and* f *such that* $\emptyset; e; \emptyset \rightarrow^* s; v; f$.

6.2 Soundness of Intensional Effect Polymorphism

The essence of intensional effect polymorphism lies in the fact that through intensional inspection (dynamic typing at the **assuming** expression), every instance of evaluation of the $\mathbb{SAFE}$ e_0 e_1 expression in the reduction sequence must be "safe," where "safety" is defined through the $\mathbb{R}$ relation concretized by the client language. To be more concrete:

▶ **Definition 11** (Effect-based Soundness of Intensional Effect Polymorphism)**.** We say e is effect-sound iff for any redex configuration such that $\mathrm{e} \trianglerighteq \lessdot s, \mathrm{e}', f \gtrdot$ and $\mathrm{e}' = \mathbb{SAFE}\ \mathrm{e}_0\ \mathrm{e}_1$, it must hold that $s; \emptyset; \emptyset \vdash_{\mathrm{D}} \mathrm{e}_0 : \mathrm{T}_0, \sigma_0$ and $s; \emptyset; \emptyset \vdash_{\mathrm{D}} \mathrm{e}_1 : \mathrm{T}_1, \sigma_1$ and $\sigma_0\ \mathbb{R}\ \sigma_1$.

Effect-based soundness is a corollary of type soundness:

▶ **Corollary 12** (λ_{ie} Effect-based Soundness). *If $\emptyset; \emptyset \vdash e : T, \sigma$, then e is effect-sound.*

There remains a gap between this property and what one *intuitively* believes the execution of $\mathbb{SAFE}$ e_0 e_1 is "safe": ultimately, what we hope to enforce is at runtime, the "monitored effect" – *i.e.*, the trace through the evaluation of e_0 and that of e_1 – does not violate what $\mathbb{R}$ represents. The definition above falls short because it relies on the dynamic typing of e_0 and e_1. To rigorously define the more intuitive notion of soundness, let us first introduce a trace-based relation induced from $\mathbb{R}$:

▶ **Definition 13** (Induced Trace Relation). $\mathbb{R}^{TR}$ is a binary relation defined over traces. We say $\mathbb{R}^{TR}$ is induced from $\mathbb{R}$ under store s iff $\mathbb{R}^{TR}$ is the smallest relation such that if $\sigma_1 \ \mathbb{R} \ \sigma_2$, then $f_1 \ \mathbb{R}^{TR} f_2$ where $s \vdash f_1 : \sigma_1$ and $s \vdash f_2 : \sigma_2$.

One basic property of our reduction system is the trace sequence is monotonically increasing:

▶ **Lemma 14** (Monotone Traces). *If $s; e; f \rightarrow s'; e'; f'$, then $f' = f, f''$ for some f''.*

Given this, we can now define the more intuitive flavor of soundness over traces:

▶ **Definition 15** (Trace-based Soundness of Intensional Effect Polymorphism). We say e is trace-sound iff for any redex configuration such that $e \trianglerighteq \langle s, e', f \rangle$ and $e' = \mathbb{SAFE}\ e_0\ e_1$, it must hold that for any s_0, e'_0, and f_0 where $s; e_0; f \rightarrow^* s_0; e'_0; f, f_0$ and any s_1, e'_1, and f_1 where $s; e_1; f \rightarrow^* s_1; e'_1; f, f_1$, then condition $f_0 \ \mathbb{R}^{TR} f_1$ holds.

To prove trace-based soundness, the crucial property we establish is:

▶ **Lemma 16** (Effect-Trace Consistency Preservation). *If $s; \Phi; \Gamma \vdash_D e : T, \sigma$, $s \vdash f : \sigma$ and $s; e; f \rightarrow s'; e'; f'$ then $s' \vdash f' : \sigma'$.*

Finally, we can prove the intuitive notion of soundness of intensional effect polymorphism:

▶ **Theorem 17** (λ_{ie} Trace-Based Soundness). *If $\emptyset; \emptyset \vdash e : T, \sigma$, then e is trace-sound.*

6.3 Differential Alignment Optimization

In Section 5, we defined an alternative "optimized λ_{ie}" to avoid full-fledged dynamic typing, centering on differential alignment. We now answer several important questions: (1) *static completeness*: every typable program in λ_{ie} has a corresponding program in optimized λ_{ie}. (2) *dynamic completeness*: for every typable program in λ_{ie}, its corresponding program at runtime cannot get stuck due to the failure of finding a differential alignment. (3) *soundness*: for every program in λ_{ie}, its corresponding program in optimized λ_{ie} should behave "predictably" at runtime. We will rigorously define this notion shortly; intuitively, it means that "optimized λ_{ie}" is indeed an *optimization* of λ_{ie}, *i.e.*, without altering the results computed by the latter.

Optimization static completeness is a simple property of $\overset{\Phi,\Gamma}{\rightsquigarrow}$:

▶ **Theorem 18** (Static Completeness of Optimization). *For any e such that $\Phi; \Gamma \vdash e : T, \sigma$, there exists d such that $e \overset{\Phi,\Gamma}{\rightsquigarrow} d$.*

To correlate the dynamic behaviors of λ_{ie} and optimized λ_{ie}, first recall that the $\rightarrow$ reduction system and $\rightarrow_O$ reduction system are identical, except for how the **assuming** expression is reduced. The progress of *(Oasm)* relies on the existence of substitution θ that aligns the dynamic type associated with values and the static type. Dynamic completeness of differential alignment thus can be viewed as the "correspondence of progress" for the two reduction systems to reduce the corresponding **assuming** expressions. This is indeed the case, which can be generally captured by the following theorem:

▶ **Theorem 19** (Dynamic Completeness of Optimization)**.** *If* $s; \Phi; \Gamma \vdash_D e : T, \sigma$ *and* $e \overset{\emptyset,\emptyset}{\rightsquigarrow} d$*, then given some* s *and* f*, the following two are equivalent:*

- *there exists some* s'*,* e' *and* f' *such that* $s; e; f \rightarrow s'; e', f'$*.*
- *there exists some* s''*,* d' *and* f'' *such that* $s; d; f \rightarrow_O s''; d', f''$*.*

Finally, we wish to study soundness. The most important insight is that the transformation relation $\overset{\Phi,\Gamma}{\rightsquigarrow}$ can be preserved through the corresponding reductions of λ_{ie} and optimized λ_{ie}. In other words, one can view the reduction of optimized λ_{ie} as a *simulation* of λ_{ie}:

▶ **Lemma 20** ($\rightarrow_O$ Simulates $\rightarrow$ with $\overset{\Phi,\Gamma}{\rightsquigarrow}$ Preservation)**.** *If* $s; \Phi; \Gamma \vdash_D e : T, \sigma$ *and* $e \overset{\emptyset,\emptyset}{\rightsquigarrow} d$ *and* $s; e; f \rightarrow s'; e', f'$ *and* $s; d; f \rightarrow_O s''; d', f''$*, then* $s' = s''$*, and* $f' = f''$*, and* $e' \overset{\emptyset,\emptyset}{\rightsquigarrow} d'$*.*

Finally, let us state our soundness of differential alignment:

▶ **Theorem 21** (Soundness of Optimization)**.** *Given some expression* e *such that* $\emptyset; \emptyset \vdash e : T, \sigma$*, and* $e \overset{\emptyset,\emptyset}{\rightsquigarrow} d$ *then*

- *there exists a reduction sequence such that* $\emptyset; e; \emptyset \rightarrow^* s; v; f$ *iff there exists a reduction sequence such that* $\emptyset; d; \emptyset \rightarrow_O^* s; v; f$*.*
- *there exists a reduction sequence such that the evaluation of* e *diverges according to* $\rightarrow$ *iff there exists a reduction sequence such that the evaluation of* d *diverges according to* $\rightarrow_O$*.*

Observe that we are careful by not stating the two reduction systems must diverge at the same time, or reduce to the same value at the same time. That would be unrealistic if the client instantiations of our calculus introduce non-determinism.

7 Related Work

Static type-and-effect systems are well-explored. Earlier work includes Lucassen [27], and Talpin *et al.* [36], and more recent examples such as Marino *et al.* [28], Task Types [21], Bocchino *et al.* [8] and Rytz *et al.* [32]. There are well-known language design ideas to improve the precision and expressiveness of static type systems, and many may potentially be applied to effect reasoning, such as flow-sensitive types [15], typestates [35] and conditional types [4]. Classic program analysis techniques such as polymorphic type inference, nCFA [33], CPA [3], context-sensitive, flow-sensitive, and path-sensitive analyses, are good candidates for effect reasoning of programs written in existing languages. For example, effect systems can gain more precision by incorporating control flow analysis (nCFA) [33] which provides precise call-site information [23].

Bañados *et al.* [5] developed a gradual effect (GE) type system based on gradual typing [34], by extending Marino *et al.* [28] with `?` ("unknown") types. As a gradual typing system, GE excels in scenarios such as prototyping. The system is also unique in its insight by viewing `?` type concretization as an abstract interpretation problem. Our work shares the high-level philosophy of GE – mixing static typing and dynamic typing for effect reasoning –

but the two systems are orthogonal in approaches. For example, GE programs may run into runtime type errors, whereas our programs do not. Foundationally, the power of intensional effect polymorphism lies upon how parametric polymorphism and intensional type analysis interact – a System F framework on the famous lambda cube – whereas frameworks based on gradual typing are not. Other than gradual typing, other solutions to mix static typing and dynamic typing include the `Dynamic` type [1], soft typing [10] and Hybrid Type Checking [14]. From the perspective of the lambda cube, their expressiveness is on par with gradual typing. Previous work, *e.g.*, Heumann *et al.* [20] and Treichler *et al.* [38], relies on dynamic effects for safe concurrency. Our system is more general: it can not only support safe concurrency as shown in Section 2.1, but also other important application domains such as information security, consistent UI access and program optimization.

Intensional type analysis by Harper and Morrisett [19] is a framework with many extensions (*e.g.*, [12]). We apply it in the context of effect reasoning, and the intentionality in our system is achieved through dynamic typing, instead of `typecase`-style inspection on polymorphic types. To the best of our knowledge, our system is the first hybrid effect type system built on top of the intensional type analysis.

Existential types are commonly used for type abstraction and information hiding. They are also suggested [19, 31] to capture the notion of `Dynamic` type [1]. Our use of existential types are closer to the latter application, except that we aim to differentiate (and connect) the types at compile time and the types at runtime, instead of pessimistically viewing the former as `Dynamic`. We are unaware of the use of *bounded* existential types to connect the two type representations.

Effect systems are an important reasoning aid with many applications. For example, beyond the application domains we described in Section 2, they are also known to be useful for safe dynamic updating [29] and checked exceptions [24, 6].

8 Conclusion

In this paper, we develop a new foundation for type-and-effect systems, where static effect reasoning is coupled with intensional effect analysis powered by dynamic typing. We describe how a precise, sound, and efficient hybrid reasoning system can be constructed, and demonstrate its applications in concurrent programming, information security, UI access, and memoization.

Acknowledgement. We would like to thank all the anonymous reviewers for their insightful comments. We thank Mehdi Bagherzadeh and members of the Panini language team for comments on initial version of this draft and discussion. Yuheng Long would like to thank the members of his PhD POS committee for constructive comments and suggestions.

References

1. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the Symposium on Principles of Programming Languages*, 1989.
2. 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, 2006.
3. Ole Agesen. *Concrete type inference: delivering object-oriented applications.* PhD thesis, Stanford University, Stanford, CA, USA, 1996.

4 Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, 1994.

5 Felipe Bañados, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming*, 2014.

6 Nick Benton and Peter Buchlovsky. Semantics of an effect analysis for exceptions. In *Proceedings of the international workshop on Types in languages design and implementation*, 2007.

7 Guy E. Blelloch. Prefix sums and their applications.

8 Robert L. Bocchino and Vikram S. Adve. Types, regions, and effects for safe programming with object-oriented parallel frameworks. In *Proceedings of the 25th European Conference on Object-oriented Programming*, 2011.

9 Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. Two for the price of one: A model for parallel and incremental computation. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, 2011.

10 Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1991.

11 Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *Proceedings of Conference on Programming Language Design and Implementation*, 2009.

12 Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *Proceedings of the International Conference on Functional Programming*, 1998.

13 Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1982.

14 Cormac Flanagan. Hybrid type checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.

15 Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

16 Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. JavaUI: Effects for controlling UI object access. In *the European Conference on Object-Oriented Programming*, 2013.

17 Aaron Greenhouse and John Boyland. An object-oriented effects system. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, 1999.

18 Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2012.

19 Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the Symposium on Principles of Programming Languages*, 1995.

20 Stephen T. Heumann, Vikram S. Adve, and Shengjie Wang. The tasks with effects model for safe concurrency. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2013.

21 Aditya Kulkarni, Yu David Liu, and Scott F. Smith. Task types for pervasive atomicity. In *the conference on Object-oriented programming, systems, languages, and applications*, 2010.

22 B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. *SIGPLAN Not.*, 12(2):1–79, 1977.

23 Anatole Le, Ondřej Lhoták, and Laurie Hendren. Using inter-procedural side-effect information in JIT optimizations. In *the International Conference on Compiler Construction*, 2005.

24 Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2), 2000.

25 Yuheng Long, Yu David Liu, and Hridesh Rajan. Hybrid Effect Checking. Technical report, Iowa State University, 2014.

26 Yuheng Long, Sean L. Mooney, Tyler Sondag, and Hridesh Rajan. Implicit invocation meets safe, implicit concurrency. In *Ninth International Conference on Generative Programming and Component Engineering*, 2010.

27 J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1988.

28 Daniel Marino and Todd Millstein. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation*, 2009.

29 Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2008.

30 Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

31 Andreas Rossberg. Dynamic translucency with abstraction kinds and higher-order coercions. *Electron. Notes Theor. Comput. Sci.*, 218:313–336, 2008.

32 Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, 2012.

33 Olin Grigsby Shivers. *Control-flow Analysis of Higher-order Languages or Taming Lambda.* PhD thesis, Carnegie Mellon University, 1991.

34 Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, 2007.

35 R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1), 1986.

36 Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2), 1994.

37 Mads Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1), 1990.

38 Sean Treichler, Michael Bauer, and Alex Aiken. Language support for dynamic, hierarchical data partitioning. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, 2013.

Type Inference for Place-Oblivious Objects

Riyaz Haque and Jens Palsberg

University of California – Los Angeles (UCLA), USA
`{rfhaque,palsberg}@cs.ucla.edu`

Abstract

In a distributed system, access to local data is much faster than access to remote data. As a help to programmers, some languages require every access to be local. A program in those languages can access remote data via first a shift of the place of computation and then a local access. To enforce this discipline, researchers have presented type systems that determine whether every access is local and every place shift is appropriate. However, those type systems fall short of handling a common programming pattern that we call place-oblivious objects. Such objects safely access other objects without knowledge of their place. In response, we present the first type system for place-oblivious objects along with an efficient inference algorithm and a proof that inference is P-complete. Our example language extends the Abadi-Cardelli object calculus with place shift and existential types, and our implementation has inferred types for some microbenchmarks.

1998 ACM Subject Classification D.3.1 Formal Definitions and Theory

Keywords and phrases parallelism, locality, types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.371

1 Introduction

Places. A distributed system consists of multiple *places* of computation. At each place, a computation may store references to both local and remote data. Access to local data is much faster than access to remote data because a remote access may go across a network. Distributed languages largely agree on the syntax of local access while they differ on the syntax of remote access. Some distributed languages, such as Titanium [26, 12], use a uniform access syntax that works for both local and remote access. Such syntax is succinct yet can make run-time performance unpredictable when the programmer is uncertain about the location of data. Other languages, such as X10 [23, 6], require a remote access to be expressed as a place shift followed by a local access at the new place. The use of place shift is verbose yet enables a programmer to easily spot slow, remote data accesses, and enables a compiler to optimize local accesses. In this paper we study a core calculus with explicit place shift.

Place checks. Languages with explicit place shift require every access to be local. This can be enforced with a run-time check known as a *place check*. The place check compares the current place with the place of the accessed data. If those two places are equal, then computation proceeds normally, and otherwise the result is a run-time error. For example, if a place check fails in X10, then the X10 implementation throws a run-time exception called BadPlaceException. Place checks can degrade the overall run-time performance [5] and they defer discovery of "place bugs" until such bugs happen at run time. However, place checking can also be done statically. For example, researchers have presented static analyses [2] and type systems [16, 5, 11, 4, 3, 15, 7, 24, 13] that determine whether every access is local. This has the potential to give programmers the best of both worlds: predictable performance and

© Riyaz Haque and Jens Palsberg;
licensed under Creative Commons License CC-BY

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 371–395

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

no place-check errors. Additionally, when a static technique can guarantee that a place check succeeds, an implementation can eliminate the run-time place check and thereby improve performance. Intuitively, the difference between static analysis and type checking in this context lies in their ambition levels. A static analysis tries to eliminate as many run-time place checks as possible, while a type system tries to eliminate *all* run-time place checks. In this paper we focus on type systems.

The challenge. We have identified a common programming pattern that we call *place-oblivious objects*. Such objects safely access other objects without knowledge of their place. We found uses of place-oblivious objects in 11 of the 13 X10 benchmarks that were considered by Lee and Palsberg [14].

Let us consider the following example written in a variant of Featherweight X10 [14]:

```
class Example {
   public Unit f;
   ...
   public void m() {
      final Unit x = this.f;
      async(x.location) {
           final O1 e = x.g;
      }
   }
}
```

We assume that `Unit` is a class with a field `g` of type `O1`. Objects of class `Example` are place oblivious. The reason is that the field `f` may at one time reference an object at place 1 and at another time reference an object at place 2. Still, method `m` successfully accesses the field `g` of objects in `f`, as follows. First, the body uses `final Unit x = this.f` to create an immutable reference `x`. This avoids trouble with any concurrent access that may change the contents of `f`. Second, the expression does a place shift `at(x.location)`, and finally a local access `x.g`.

In this paper, we will use an extension of the Abadi-Cardelli object calculus [1] rather than X10. We chose the Abadi-Cardelli object calculus because it has a succint semantics and a small number of type rules, which makes it a good basis for study of algorithms and for detailed proofs. In our calculus, we can write an expression similar to the body of `m` in the following way:

$$\mathit{open}\ x = \mathit{this}.f\ \mathit{in}\ \mathit{at}(x.\mathit{place})\{\ x.g\ \}$$

First, the expression uses *open*, which has the same semantics as *let* but which we will give a different type rule. The *open* expression creates an immutable reference $x = \mathit{this}.f$ (like `final` does in the X10 code above), then does a place shift $\mathit{at}(x.\mathit{place})$, and finally a local access $x.g$.

The type system must *(i)* assign f a type that is compatible with the types of the objects at place 1 and place 2, and *(ii)* determine that after the place shift, $x.g$ is indeed a local access. We cannot simply give f a type that says that the place of f is unknown. Such a type would imply that the type system has no knowledge of the target of the place shift, hence no basis for knowing whether $x.g$ is a local access. Place-oblivious objects occur frequently in X10 code and yet previous work falls short of type inference for such objects, mainly because of a lack of a sensible type for f.

Our results. We present the first type system for place-oblivious objects along with an efficient inference algorithm and a proof that inference is P-complete. Our type system is sound: a well-typed program is *place safe*, that is, every access is local. Our example language extends the Abadi-Cardelli object calculus [1] with place shift and existential types, and has no type annotations. Every value is an object, and every object resides permanently at a specific place. Places surface only during place shift. Our implementation has successfully inferred types for some microbenchmarks.

Our type system. We have two forms of type, namely (1) a pair of a usual object type and a place type, and (2) a *packed type*, which is just an object type. We give a packed type to a term whose evaluation potentially creates objects at different places. The idea of a packed type is to "forget" the place type. Intuitively, a packed type is a light-weight existential type [21, 22, 19]:

$$\exists\pi.(\text{object type}, \pi)$$

where the place type π cannot occur free in the object type. In contrast to most calculi with existential types, our calculus has no type annotations. In particular, we introduce a packed type via implicit *subtyping* from a usual type to a packed type, rather than with an explicit "pack" operation. Additionally, we eliminate a packed type via the an *open* construct. In the example above, we would use subtyping to give the objects at place 1 and place 2 the same packed type, and we would give f that packed type, too. Then the *open* construct assigns a fresh place Skolem constant X to x, and finally we can successfully type check $at(x.place)\{\ x.g\ \}$ because the two occurrences of x have the same place type X.

Our inference algorithm. Our type inference algorithm is the first polynomial-time inference algorithm for existential types of which we are aware. The main technical challenge for type inference is to handle the type rule for the *open* construct. That rule introduces a place Skolem constant X for the unknown place of an object with a packed type, it assigns a type B to code that uses the packed object, and finally it requires $X \notin \Delta$ and $FV(B) \subseteq \Delta$, where Δ is a list of place Skolem constants used in enclosing *open* constructs. We handle those conditions with a novel technique. Our inference algorithm has two steps.

The first step of our algorithm transforms the type inference problem to a constraint-satisfiability problem. We show that those two problems are equivalent. Each constraint is of one of these five forms:

$$u \leq_O v \qquad O \subseteq_O K \qquad H \leq_H H \qquad H \in_H K \qquad H \neq_H \mathsf{unkn}$$

where u, v are type expressions, O is an object-type variable, K is a finite set of place types, and H is a place type variable that must be assigned a place or unkn. Intuitively, $\leq_O$ denotes subtyping, $\subseteq_O$ denotes that the type denoted by O uses only places in K, $\leq_H$ denotes subtyping for place types, $\in_H$ denotes set membership, and $\neq_H$ denotes inequality. Palsberg showed in 1993 how to solve constraints of the form $u \leq_O v$ in $O(n^3)$ time [20]. The main new challenge are the constraints of the form $O \subseteq_O K$. The problem is that a solution may assign O a deeply nested type and we need to know that every level uses only places in K.

The second step of our algorithm performs a solution-preserving *closure* of the constraint set. We show that a closed constraint set is satisfiable if and only if it is *well formed* and *consistent*. We define closure with a novel set of Horn clauses, we define *well formed* to mean that each constraint of the form $u \leq_O v$ has no top-level violation of subtyping, and we define *consistent* to mean that the constraints of the forms $H \in_H K$ and $H \neq_H \mathsf{unkn}$ have no

obvious inconsistencies. The most time-consuming steps are the closure and the consistency check, which both take $O(n^3)$ time. In total, our algorithm takes $O(n^3)$ time.

The rest of the paper. In Section 2 we present our calculus and type system, and in Section 3 we discuss sixteen examples. In Section 4 we show that type inference is equivalent to a constraint-satisfiability problem, in Section 5 we present our type inference algorithm along with an example of how type inference works, in Section 6 we give further discussion of our results, and in Section 7 we give a detailed comparison with related work. Our paper states seven theorems; we prove one of them in the main body of the paper, five of them in the appendices of the full version of the paper, which is available from our website [9], while we leave one straightforward proof to the reader.

2 Our Language

We now present the syntax, operational semantics, and type system for our calculus.

Syntax. Here is the grammar for terms:

$$
\begin{array}{rcll}
a, b, c & := & s, x, y & \text{(variables)} \\
& | & o & \text{(object)} \\
& | & a.l & \text{(method call)} \\
& | & a.l \Leftarrow \varsigma(x)\ b & \text{(method update)} \\
& | & at(a.place)\ b & \text{(at } a\text{'s place)} \\
& | & at(\rho)\ b & \text{(at place } \rho) \\
& | & open\ x = a\ in\ b & \text{(let-binding)} \\
o & := & [l_i = \varsigma(x_i)\ b_i\ ^{i \in 1..n}] & \text{(object, } l_i \text{ distinct)} \\
v & := & at(\rho)\ o & \text{(value)} \\
\rho & \in & \mathsf{Places} & \text{(place constant)}
\end{array}
$$

The first four productions are those of the Abadi-Cardelli object calculus [1]. Those four productions enable us to use variables, create objects, and do method call and method update. An object defines n methods named l_i, for i between 1 and n. In a method definition $\varsigma(x_i)b$, the binder ς binds a variable x_i which refers to the entire object, in analogy with *self* in Smalltalk and *this* in Java. Additionally, b is the body of the method; the method returns the value of b. In a method call $a.l$, the callee is the method l in object a. A method update $a.l \Leftarrow \varsigma(x)\ b$, replaces the method l in object a with method $\varsigma(x)b$.

Notice that methods have no parameters, aside from a name for the receiver object, so a method type is only about a return value, not parameters. Additionally, methods are written with a ς rather than a λ, and they can be updated, which means they can also work as fields, Abadi and Cardelli showed how to encode the λ-calculus into their calculus.

The last three productions provide our extension of the Abadi-Cardelli calculus. The fifth and sixth productions enable place shift, either to the place of an object a or to a place constant ρ from a finite set Places. The last production enables us to open an object, i.e., to abstract the place of an object.

Operational semantics. Figure 1 shows the small-step operational semantics. Every value is of the form $at(\rho)\ o$, which is an object at place ρ.

We use the notation $b[x := a]$ to denote the application of a substitution $[x := a]$ to b. As usual, $b[x := a]$ denotes b with every free occurrence of x replaced with a, assuming that (if needed) all local names in b have been renamed to avoid clashes with free names in a.

Term reduction judgment: $\rho \vdash a \rightarrow a'$

(O-Obj) $$\frac{}{\rho \vdash o \rightarrow at(\rho)\ o}$$

(O-Call-Cong) $$\frac{\rho \vdash a \rightarrow a'}{\rho \vdash a.l \rightarrow a'.l}$$

(O-Call-Comp) $$\frac{o \equiv [l_i = \varsigma(x_i)\ b_i\ ^{i \in 1..n}] \quad j \in 1..n \quad \rho = \rho'}{\rho \vdash (at(\rho')\ o).l_j \rightarrow b_j[x_j := at(\rho')\ o]}$$

(O-Update-Cong) $$\frac{\rho \vdash a \rightarrow a'}{\rho \vdash a.l \Leftarrow \varsigma(x)\ b \rightarrow a'.l \Leftarrow \varsigma(x)\ b}$$

(O-Update-Comp) $$\frac{\begin{array}{l} o \equiv [l_i = \varsigma(x_i)\ b_i\ ^{i \in 1..n}] \\ o' \equiv [l_i = \varsigma(x_i)\ b_i\ ^{i \in 1..n/\{j\}},\ l_j = \varsigma(x)\ b] \\ j \in 1..n \quad \rho = \rho' \end{array}}{\rho \vdash (at(\rho')\ o).l_j \Leftarrow \varsigma(x)\ b \rightarrow at(\rho')\ o'}$$

(O-AtObject-Cong) $$\frac{\rho \vdash a \rightarrow a'}{\rho \vdash at(a.place)\ b \rightarrow at(a'.place)\ b}$$

(O-AtObject-Comp) $$\frac{}{\rho \vdash at((at(\rho')\ o).place)\ b \rightarrow at(\rho')\ b}$$

(O-AtConst-Cong) $$\frac{\rho' \vdash b \rightarrow b'}{\rho \vdash at(\rho')\ b \rightarrow at(\rho')\ b'}$$

(O-AtConst-Ret) $$\frac{}{\rho \vdash at(\rho')\ v \rightarrow v}$$

(O-Open-Cong) $$\frac{\rho \vdash a \rightarrow a'}{\rho \vdash open\ x = a\ in\ b \rightarrow open\ x = a'\ in\ b}$$

(O-Open-Comp) $$\frac{o \equiv [l_i = \varsigma(x_i)\ b_i\ ^{i \in 1..n}]}{\rho \vdash open\ x = at(\rho')\ o\ in\ b \rightarrow b[x := at(\rho')\ o]}$$

Figure 1 Operational semantics.

We use the judgment $\rho \vdash a \rightarrow a'$ to denote that at place ρ, the term a takes a step to term a'. The first five rules are variants of rules for the Abadi-Cardelli calculus. The differences are the addition of a place to each judgment and the conditions $\rho = \rho'$ in rules (S-Call-Comp) and (S-Update-Comp). Those conditions express place checks that an implementation must perform at every method call and every method update. For example, in rule (S-Call-Comp), the place check says that if we want to call a method in an object at place ρ', then we need ρ' to equal the current place ρ. In other words, the access is local. If the place check fails, then the method call or method update is stuck.

Notice that we could have written (S-Call-Comp) and (S-Update-Comp) in a simpler way by replacing ρ' with ρ and omitting the explicit condition $\rho = \rho'$. We prefer the explicit style that emphasizes the place check.

The next four rules express the semantics of place shift. In particular, rule (S-AtObject-Comp) expresses that the place of $at(\rho')o$ is ρ', while rule (S-AtConst-Ret) expresses that if the body of a place shift has evaluated to a value, then we can return that value.

The final two rules express the semantics of *open*, which is the same as the semantics of standard *let*-binding. We will use a different type rule for *open* than the usual one for *let*.

We write $\rho \vdash a \to^* a'$ if either $a = a'$, or $\rho \vdash a \to^* a''$ and $\rho \vdash a'' \to a'$.

We say that a term a is *stuck* at place ρ if a is not a value and a cannot use the rules in Figure 1 to take a step at ρ. We say that a term a can *go wrong* at place ρ if for some a', we have $\rho \vdash a \to^* a'$ and a' is stuck at ρ.

Notice that our notion of going wrong embodies the dual of a notion of *place safety*, which means that every access is local. The reason is that the semantics does place checks that leave the execution stuck if a check fails. Thus, if a term a cannot go wrong at place ρ, then we can know that every place check succeeds; hence that every access is local.

Type system. The goal of our type system is to guarantee that well-typed programs cannot go wrong. In particular, we want well-typed programs to be place safe. Accordingly, our type system has a static place check for each case where the semantics has a run-time place check. If a static place check fails, the result is a type error. Here is the grammar for types:

$$\begin{array}{rcll} A, B & := & ([l_i : B_i \ ^{i \in 1..n}], \pi) & \text{(locality type)} \\ \pi & := & X, Y & \text{(place Skolem constant)} \\ & \mid & \mathsf{unkn} & \text{(packed place)} \\ & \mid & \rho & \text{(place type constant)} \end{array}$$

As shown above, a type in our system is a pair; the first part is the object type analogous to the standard Abadi-Cardelli object type and the second part is the place type π denoting the place where an object resides. Notice that an object type can be [ ] (that is, empty), which happens when $n = 0$. A place type can either be a constant (statically known), a place Skolem constant (statically unknown but immutable) or the special type unkn (unknown). Intuitively, unkn says that there exists some place where the object resides. We use the *open* construct in our calculus to convert this existential quantification into a place Skolem constant.

Give a type $A \equiv ([l_i : B_i \ ^{i \in 1..n}], \pi)$, we define its *object component* as $obj(A) = [l_i : B_i \ ^{i \in 1..n}]$ and its *place component* as $pl(A) = \pi$.

Place Skolem constants and unknown places. We assume that place Skolem constants are drawn from a countable set $\mathsf{Skolems}$. We introduce the constant unkn where $\mathsf{unkn} \notin \mathsf{Skolems}$. We will use the syntactic sugar

$$packed\ [l_i : B_i \ ^{i \in 1..n}] = ([l_i : B_i \ ^{i \in 1..n}], \mathsf{unkn})$$

which enables simpler definitions in the following.

Figure 2 shows three well-formedness rules, three subtyping rules, and nine type assignment rules. First we explain the well-formedness rules. Rule (W-Place), Rule (W-Type-Obj), and Rule (W-Env) ensure, respectively, that a place type, a locality type and the environment are well-formed with respect to a set of place Skolem constants Δ. Here, $FV(A) \subseteq \Delta$, where $A \equiv ([l_i : B_i \ ^{i \in 1..n}], \pi)$, means that

Well-formedness:

$$(\text{W-Place})\ \frac{FV(\pi) \subseteq \Delta}{\Delta \vdash_p \pi} \qquad (\text{W-Type-Obj})\ \frac{FV(A) \subseteq \Delta}{\Delta \vdash_T A} \qquad (\text{W-Env})\ \frac{\Delta \vdash_T \Gamma(x) \quad \forall x \in dom(\Gamma)}{\Delta \vdash_E \Gamma}$$

Subtyping:

$$(\text{S-Ident})\ \frac{}{A \leq A} \qquad (\text{S-Trans})\ \frac{A \leq B \quad B \leq C}{A \leq C}$$

$$(\text{S-Obj})\ \frac{\pi \leq \pi'}{([l_i : B_i\ ^{i \in 1..n+k}], \pi) \leq ([l_i : B_i\ ^{i \in 1..n}], \pi')}$$

Type assignment:

$$(\text{T-Sub})\ \frac{\Delta; \Gamma; \pi_c \vdash a : A \quad A \leq B}{\Delta; \Gamma; \pi_c \vdash a : B} \qquad (\text{T-Var})\ \frac{\Gamma(x) = B}{\Delta; \Gamma; \pi_c \vdash x : B}$$

$$(\text{T-Obj})\ \frac{\forall j \in 1..n \ \ \Delta; (\Gamma, x_j : A); \pi_c \vdash b_j : B_j \quad \Delta \vdash_T A \quad A \equiv ([l_i : B_i\ ^{i \in 1..n}], \pi_c)}{\Delta; \Gamma; \pi_c \vdash [l_i = \varsigma(x_i)\ b_i\ ^{i \in 1..n}] : A}$$

$$(\text{T-Call})\ \frac{\Delta; \Gamma; \pi_c \vdash a : A \quad j \in 1..n \quad A \equiv ([l_i : B_i\ ^{i \in 1..n}], \pi) \quad \pi_c = \pi}{\Delta; \Gamma; \pi_c \vdash a.l_j : B_j}$$

$$(\text{T-Update})\ \frac{\begin{array}{c}\Delta; \Gamma; \pi_c \vdash a : A \quad \Delta; (\Gamma, x : A); \pi \vdash b : B_j \quad j \in 1..n \\ A \equiv ([l_i : B_i\ ^{i \in 1..n}], \pi) \quad \pi_c = \pi\end{array}}{\Delta; \Gamma; \pi_c \vdash a.l_j \Leftarrow \varsigma(x)\ b : A}$$

$$(\text{T-AtObject})\ \frac{\Delta; \Gamma; \pi_c \vdash a : A \quad A \equiv ([l_i : B_i\ ^{i \in 1..n}], \pi) \quad \Delta; \Gamma; \pi \vdash b : B}{\Delta; \Gamma; \pi_c \vdash at(a.place)\ b : B}$$

$$(\text{T-AtConst})\ \frac{\Delta; \Gamma; \rho \vdash b : B}{\Delta; \Gamma; \pi_c \vdash at(\rho)\ b : B}$$

$$(\text{T-Open})\ \frac{\begin{array}{c}\Delta; \Gamma; \pi_c \vdash a : A \quad (\Delta, X); (\Gamma, x : (obj(A), X)); \pi_c \vdash b : B \\ X \notin \Delta \quad \Delta \vdash_T B\end{array}}{\Delta; \Gamma; \pi_c \vdash open\ x = a\ in\ b : B}$$

$$(\text{T-Prog})\ \frac{\Delta \vdash_E \Gamma \quad \Delta \vdash_p \pi_c \quad \Delta; \Gamma; \pi_c \vdash a : A}{\vdash_P (\Delta, \Gamma, \pi_c, a, A)}$$

Figure 2 Type rules.

$\Delta \vdash_p \pi \ \wedge \ \forall i \in 1..n : FV(B_i) \subseteq \Delta.$
Note that for every Δ, we have $\Delta \vdash_p$ unkn.

Next we explain the subtyping rules. Those rules rely on this definition of "width" subtyping for object types:

$$[l_i : B_i\ ^{i \in 1..n+k}] \leq [l_i : B_i\ ^{i \in 1..n}]\ .$$

Additionally, the subtyping rules rely on this definition of subtyping between place types:

$\pi \leq \pi \quad \pi \leq \mathsf{unkn} \quad \mathsf{unkn} \leq \mathsf{unkn}$

Notice that $\pi \leq \mathsf{unkn}$ can help establish a subtyping relationship between a locality type and its packed form, which we use to mask the place of an object. Rule (S-Ident) and Rule (S-Trans) are standard reflexivity and transitivity rules, while Rule (S-Obj) combines "width" subtyping rule for object types with subtyping for place types.

Finally we explain the type assignment rules. We use the judgment $\Delta; \Gamma; \pi_c \vdash a : A$ to denote that under the context $\Delta; \Gamma; \pi_c$, the term a has type A. In the context $\Delta; \Gamma; \pi_c$, we use Δ to denote a list of place Skolem constants, and we use Γ to denote a finite map from variables to types. We refer to π_c as the *place context*, that is, the type of the current place of execution.

The first rule Rule (T-Sub) is the standard subtyping rule. The next four type rules are variants of the type rules for the first-order type system for the Abadi-Cardelli object calculus. The main difference is in rules (T-Call) and (T-Update) that each contains the condition $\pi_c = \pi$, which is a type-level place check. The idea is that if the type-level place check succeeds, then the term-level place-check succeeds, too. For example, in rule (T-Call) the place check $\pi_c = \pi$ says that if we want to call a method in an object at a place with type π then we need π to equal the type π_c of the current place. In other words, the access is local. If the place check fails, then the program won't type check. Also, Rule (T-Obj) contains the check $\Delta \vdash_T A$ to ensure that the resulting object type is well formed under Δ.

The next two rules type check place shift. In both cases, the type of current place is π_c yet shifts to be π in rule (T-AtObject) or ρ in rule (T-AtConst).

Rule (T-Open) is a simplified version of the corresponding rule for full-blown existential types. It says that we can substitute a fresh place Skolem constant X for the (possibly unknown) place type of a as long as we ensure that X doesn't escape the type of the body b. This allows us to treat the place of a in an abstract manner. Unlike most rules for existential types, we do not require a to have a packed type. This is merely a technical convenience; we can always use subtyping (Rule (S-Obj)) to get a packed type for a.

Finally, Rule (T-Prog) ensures that the initial Γ and π_c for a term are well-formed with respect to the initial Δ.

Type soundness. We use the standard technique of preservation and progress [21, 25] to prove type soundness. As a key step, we introduce the notion of *place independence.* A term a is place independent if and only if we have that

if $\Delta; \Gamma; \pi_c \vdash a : A$, then $\forall \pi$: $\Delta; \Gamma; \pi \vdash a : A$.

We first show that values are place independent and use that to prove a standard substitution lemma, which in turn is the corner stone of the proof of preservation.

▶ **Theorem 1** (Soundness). *If $\emptyset; \emptyset; \rho \vdash a : A$, then a cannot go wrong at ρ.*

Theorem 1 says that a well-typed program cannot go wrong, hence the program is place safe: every access is local. We prove Theorem 1 in Appendix A of the full version of the paper [9].

Type inference. Let $1 \in \mathsf{Places}$ be the initial place of computation. The type inference problem is:

Given a term a, does there exist a type A such that $\emptyset; \emptyset; 1 \vdash a : A$?

We will show how to do type inference in polynomial time.

Syntactic sugar. We will use a variant of Abadi and Cardelli's encoding of let-expressions. Suppose s doesn't occur free in a, b, and define the following object o and syntactic sugar for *let*:

$$\begin{array}{rcl} o & \equiv & [f = \varsigma(s)\ a,\ r = \varsigma(s)\ b[x := at(s.place)\ s.f]] \\ let\ x = a\ in\ b & \equiv & o.r \end{array}$$

The idea of the encoding is to create an object o that facilitates the connection between a and b. We store a in field f and we store b in field r. Computation can now begin with a call to $o.r$. The substitution $b[x := at(s.place)\ s.f]$ replaces all references of x in b with accesses to $s.f$. The main novel aspect is $at(s.place)$ which ensures that $s.f$ place checks. Intuitively, we store the result of a in field f at the *current* place, yet when the expression b references x, the computation may have moved to a *different* place. The use of $at(s.place)$ makes the access happen at the place where f is.

▶ **Theorem 2** (Derived Type Rule)**.**

$$\text{(T-Let)}\quad \frac{\Delta;\Gamma;\pi_c\ \vdash\ a : A \qquad \Delta;(\Gamma, x : A);\pi_c\ \vdash\ b : B \qquad \Delta \vdash_T A, B \qquad \Delta \vdash_p \pi_c}{\Delta;\Gamma;\pi_c\ \vdash\ let\ x = a\ in\ b : B}$$

We prove Theorem 2 in Appendix B of the full version of the paper [9]. A key lemma is that $at(s.place)\ s.f$ is place independent.

3 Examples

In this section we discuss several example programs that demonstrate key properties of our calculus. In Section 3.1 we kick off with four straightforward examples. In Section 3.2 we continue with four more advanced examples that remain within what can be handled by previous work. In Section 3.3 we finally get to eight examples of place-oblivious objects. For each example, we will either show the type produced by our inference algorithm, or we will discuss why the example has no type. Later in Section 5.4, we show how type inference works for one of the advanced examples. Featherweight X10 versions of the examples are available from our website [9].

Let o_1 denote a closed value with type B_1, that is, for any Δ, Γ and π_c, we can derive $\Delta;\Gamma;\pi_c\ \vdash\ o_1 : B_1$.

3.1 Place safety with statically known places

An object can be dereferenced safely at its own (statically known) place of creation. A type system equipped with a local/non-local place analysis should be able to track this simplest form of place correlation.

▶ Example 1.

$$\begin{array}{llcl} at(1) & [\ l & = & \varsigma(s)\ at(1)\ [r = \varsigma(s')\ o_1], \\ & \ \ m & = & \varsigma(s)\ at(1)\ s.l \\ &] & & \end{array}$$

The example is place safe since the outermost object, s, is both allocated and always dereferenced at place 1. The place check in Rule (T-Call) for $s.l$ succeeds and the program type checks; s has the type $([l : \mathit{packed}\ [], m : \mathit{packed}\ []], 1)$.

▶ Example 2.

$$\begin{array}{llll} at(1) & [& l & = & \varsigma(s)\ at(1)\ [r = \varsigma(s')\ o_1], \\ & & m & = & \varsigma(s)\ at(1)\ s.l.r \\ &] & & & \end{array}$$

Compared to Example 1, we have changed the body of m from $s.l$ to $s.l.r$. This is still place safe because the object returned in the body of l is also allocated at place 1. Thus for $s.l.r$, the place check in both the uses of Rule (T-Call) succeeds and hence the program type checks; s has the type $([l : ([r : \mathit{packed}\ []], 1), m : \mathit{packed}\ []], 1)$.

▶ Example 3.

$$\begin{array}{llll} at(1) & [& l & = & \varsigma(s)\ at(1)\ [r = \varsigma(s')\ o_1], \\ & & m & = & \varsigma(s)\ at(2)\ s.l \\ &] & & & \end{array}$$

Compared to Example 1, we now change the body of m to instead execute at place 2. This fails since s, allocated at place 1, is dereferenced at place 2. Thus the place check in Rule (T-Call) for $s.l$ fails and the program does not type check.

▶ Example 4.

$$\begin{array}{llll} at(1) & [& l & = & \varsigma(s)\ at(1)\ [r = \varsigma(s')\ o_1], \\ & & m & = & \varsigma(s)\ at(2)\ at(1)\ s.l \\ &] & & & \end{array}$$

Compared to Example 3, the body of m starts execution at place 2 but immediately switches to place 1 and then evaluates $s.l$. This is place safe and the place check in Rule (T-Call) succeeds; s gets the type $([l : \mathit{packed}\ [], m : \mathit{packed}\ []], 1)$. Intuitively, $at(2)\ at(1)\ s.l$ is semantically equivalent to $at(1)\ s.l$.

3.2 Place safety that can be checked by previous work

As long it can be inferred that two objects are created at the same place, one can be dereferenced safely while executing at the other's (possibly abstract) place. A type system with a clever locality analysis can type check such programs.

▶ Example 5.

$$\begin{array}{llll} at(1) & [& l & = & \varsigma(s)\ [r = \varsigma(s')\ o_1], \\ & & m & = & \varsigma(s)\ at(s.place)\ s.l.r \\ &] & & & \end{array}$$

In this example, we allocate the body of l at the place of s. Also, in the body of m, we evaluate $s.l.r$ at the (abstract) place of s. It is valid to dereference s at its own place. Also, the access $l.r$ is place safe since the body of l is allocated at s's place. Hence the place checks in Rule (T-Call) for $s.l.r$ succeed and the program type checks; s gets the type $([l : ([r : \mathit{packed}\ []], 1), m : \mathit{packed}\ []], 1)$. Note that even though s's place is statically known (place 1), place safety analysis is actually independent of that; the program would still type check if $at(1)$ is changed to $at(0)$.

▶ Example 6.

$$\begin{array}{lllll} at(1) & [& l & = & \varsigma(s)\ [r = \varsigma(s')\ o_1], \\ & & m & = & \varsigma(s)\ at(s.l.place)\ s.p, \\ & & p & = & \varsigma(s)\ o_1 \\ &] & & & \end{array}$$

Here the body of field l is allocated at the same place as s. Hence it is place safe to access $s.p$ at l's place. This program type checks; s has the type $([l : ([], 1), m : packed\ [], p : packed\ []], 1)$.

▶ Example 7.

$$\begin{array}{lllll} at(1) & [& l & = & \varsigma(s)\ [r = \varsigma(s')\ o_1], \\ & & m & = & \varsigma(s)\ at(2)\ at(s.l.place)\ s.l.r \\ &] & & & \end{array}$$

Here we want to evaluate $s.l.r$ at l's place. Similar to Example 6, this seems valid since the body of l is allocated at s's place. However, this program fails to type check because $s.l$ in $at(s.l.place)$ is first evaluated at place 2. Since s is created at place 1, the place check fails. Notice that Example 3 fails for the same reason.

▶ Example 8.

$$\begin{array}{lllll} at(1) & [& l & = & \varsigma(s)\ [r = \varsigma(s')\ o_1], \\ & & m & = & \varsigma(s)\ let\ f = s.l\ in\ at(2)\ at(f.place)\ s.l.r \\ &] & & & \end{array}$$

The problem in Example 7 is solved by introducing a *let*-expression to first ensure that $s.l$ is evaluated at s's place (place 1). Since f, l and s are at the same place, $s.l.r$ can now be safely evaluated at f's place. This program type checks; the type of s is $([l : ([r : packed\ []], 1), m : packed\ []], 1)$.

3.3 Place safety for place-oblivious objects

In Examples 5–8, place safety is based upon the abstract place of an object and on that a field stays immutable once assigned. However, in Example 5, if we add the update $s.l \Leftarrow \varsigma(s')\ at(2)\ [r = \varsigma(s')\ o_1]$, then the program will fail since the contents of l is initially allocated at s's place (place 1). This is restrictive in cases where a field might be assigned objects from different places, such as a server receiving objects from multiple nodes. We use subtyping to mask an object's place and subsequently we use *open* to "reveal" it, and thereby we ensure that a field can be safely updated.

We now show eight examples of place-oblivious objects. First we show a program that embodies the main example in the introduction, and then we show two additional programs that adds the kind of update that we discussed in the previous paragraph. After that, we finish with five more advanced examples.

▶ Example 9.

$$\begin{array}{llll} [& l & = & \varsigma(s)\ at(1)\ [r = \varsigma(s')\ o_1], \\ & m & = & \varsigma(s)\ open\ x = s.l\ in\ at(x.place)\ x.r \\ ] & & & \end{array}$$

This example is a place-oblivious object. The body of l returns an object created at place 1. In the body of m, we first open the place of l, essentially abstracting the place of field l as

an unknown but immutable constant. We then proceed to access its field r at that place. Note that this is place safe since we are always accessing l's fields at its own place (though without actually delving into what that place is.) Hence this example type checks; s has the type $([l : packed\ [r : packed\ []], m : packed\ []], 1)$, x gets the type $([r : packed\ []], X)$ and l's body has the type $packed\ [r : packed\ []]$.

▶ Example 10.

$$\begin{array}{rcl} [\quad l &=& \varsigma(s)\ at(1)\ [r = \varsigma(s')\ o_1], \\ m &=& \varsigma(s)\ open\ x = s.l\ in\ at(x.place)\ x.r \\ ].l &\Leftarrow& \varsigma(s)\ at(2)\ [r = \varsigma(s')\ o_1] \end{array}$$

Extending Example 9, we now proceed to update the body of l with an object at place 2. This is still place safe since subtyping ensures that l is updated with a body that returns a packed object of the same type as the original method and we still access l's fields only after opening its place. Thus this example type checks; s has the type $([l : packed\ [r : packed\ []], m : packed\ []], 1)$, x gets the type $([r : packed\ []], X)$ and l's body retains the type $packed\ [r : packed\ []]$ before and after the update.

▶ Example 11.

$$\begin{array}{rcl} [\quad l &=& \varsigma(s)\ at(1)\ [r = \varsigma(s')\ o_1], \\ m &=& \varsigma(s)\ at(s.l.place)\ s.l.r \\ ].l &\Leftarrow& \varsigma(s)\ at(2)\ [r = \varsigma(s')\ o_1] \end{array}$$

This example is a variation of Example 10 in which we have inlined the definition of x. In the expression $at(s.l.place)\ s.l.r$, the method update may change the contents of $s.l$ *between* the evaluation of $s.l.place$ and the evaluation of $s.l.r$. (The semantics in Section 2 is sequential but can be changed to a more general style of reduction that supports the described behavior.) The result can be a run-time place-check error. In the example, before the update, $s.l$ contains an object at place 1, while after the change, $s.l$ contains an object at place 2. The example fails to type check. The reason is that in the absence of *open* (as in Example 10), we have no sensible type for field l. Example 10 shows how our approach uses $open\ x = s.l$ explicitly to create an immutable reference that avoids the stated problem and helps make the example type check.

▶ Example 12.

$$\begin{array}{rcl} open\ x &=& [r = \varsigma(s')\ o_1]\ in \\ open\ y &=& [r = \varsigma(s')\ o_1]\ in \\ [l &=& \varsigma(s)\ at(x.place)\ y.r] \end{array}$$

Here we open two packed objects as different variables x and y and then try to access y's field at x's place. This example is place safe since both the packed objects are created at the same place. However, this program does not type check because upon opening, both x and y are assigned different place types, say X and Y. While checking $y.r$, the place check in Rule (T-Call) fails since the type system assumes that $X \neq Y$.

▶ Example 13.

$$\begin{array}{rcl} open\ x &=& o_1\ in \\ let\ y &=& \\ [l &=& \varsigma(s)\ at(x.place)\ [r = \varsigma(s')\ o_1], \\ m &=& \varsigma(s)\ let\ f = s.l\ in\ at(x.place)\ f.r \\ ] & in & y.m \end{array}$$

This program is similar to examples 9 and 11. The only difference is that we now get an extended place context that includes a new place type (X) created by unpacking an object into x. We allocate the body of l and access its fields at X. Note that we still need the *let*-expression since s is not created at X. The example type checks; assuming execution starts at place 1, s gets the type $([l : ([r : packed\ []], X), m : packed\ []], 1)$.

▶ Example 14.

$$\begin{array}{rcl} open\ x & = & o_1\ in \\ open\ y & = & o_1\ in \\ \quad [l & = & \varsigma(s)\ at(y.place)\ [q = \varsigma(s')\ o_1], \\ m & = & \varsigma(s)\ let\ f = s.l\ in\ at(x.place)\ f.q \\ \quad] & & \end{array}$$

This program is a variation of Example 13. Here we have an extended place context with two new type variables X and Y using two *open* expressions. This example fails to type check because we try to access an object created on place Y at the place X. Note that like Example 12, this program is place safe.

▶ Example 15.

$$\begin{array}{lcl} [l & = & \varsigma(s)\ [r = \varsigma(s')\ o_1], \\ m & = & \varsigma(s)\ open\ x = s.l\ in\ x \\ ].m.r & & \end{array}$$

In this program, the body of field m opens the place of field l's object and returns the object with a new abstract place X. By Rule (T-Open), X is not visible outside the scope of the *open* expression, hence subtyping assigns a packed type to m's body. However, since an object with a packed type cannot be dereferenced, this example fails to type-check during dereferencing of field r in Rule (T-Call).

▶ Example 16.

$$\begin{array}{rcl} open\ x & = & o_1\ in \\ at(1)\ ([l & = & \varsigma(s)\ at(x.place)\ [q = \varsigma(z)\ [r = \varsigma(s')\ o_1]], \\ m & = & \varsigma(s)\ at(x.place)\ [r = \varsigma(s')\ o_1], \\ p & = & \varsigma(s)\ let\ f = s.l\ in \\ & & \qquad at(f.place)\ (f.q \Leftarrow \varsigma(s')\ at(s.place)\ s.m) \\ w & = & \varsigma(s)\ o_1 \\ ]).w & & \end{array}$$

Finally, a slightly more complicated example. The body of l is allocated at the abstract place X obtained by opening an object in variable x. As can be seen, the bodies of q and m are also allocated at place X. Thus it is place safe to update q's body with m's body at the place of l. This example type checks. We do need a *let*-expression in the body of p for the reason mentioned earlier. Here s gets the type $([l : ([q : packed\ []], X), m : packed\ [], p : packed\ []], 1)$.

4 From Types to Constraints

We show how to reduce type inference to a constraint-satisfiability problem. We define $\mathcal{K} = \mathsf{Places} \cup \mathsf{Skolems} \cup \{\mathsf{unkn}\}$.

Constraint systems. An *ACD-system* is a triple $(\mathcal{V}, \mathcal{W}, \mathcal{Q})$ where $\mathcal{V}$ is a finite set of variables (typically O) that each ranges over record types of the form $[l_i : B_i \ {}^{i\in 1..n}]$, where $\mathcal{W}$ is a finite set of variables (typically H) that each ranges over $\mathcal{K}$, and where $\mathcal{Q}$ is a finite set of constraints of the five forms:

$$u \leq_O v \qquad O \subseteq_O K \qquad H \leq_H H \qquad H \in_H K \qquad H \neq_H \mathsf{unkn}$$

where u, v are either O or $[l_i : (O_i, H_i) \ {}^{i\in 1..n}]$, and where K ranges over finite subsets of $\mathcal{K}$. We can view a constraint $H \neq_H \mathsf{unkn}$ as a readable way to write $H \in_H \mathcal{K} \setminus \{\mathsf{unkn}\}$. We overload $\mathcal{Q}$ and use $\mathcal{Q}$ to denote $(\mathcal{V}, \mathcal{W}, \mathcal{Q})$.

Suppose h is a mapping from $\mathcal{V}$ to record types of the form $[l_i : B_i \ {}^{i\in 1..n}]$, and from $\mathcal{W}$ to $\mathcal{K}$. Define $\tilde{h}$ as:

$$\tilde{h}(O) = h(O) \qquad \tilde{h}(H) = h(H) \qquad \tilde{h}([l_i : (O_i, H_i) \ {}^{i\in 1..n}]) = [l_i : (h(O_i), h(H_i)) \ {}^{i\in 1..n}]$$

Let $\tilde{h}(O) \subseteq K$ denote that for every subtree of the form $([l_i : B_i \ {}^{i\in 1..n}], \pi)$ or $([l_i : B_i \ {}^{i\in 1..n}], \mathsf{unkn})$ in the syntax tree of $\tilde{h}(O)$, $\pi \in K$ (or $\mathsf{unkn} \in K$). Given $\tilde{h}(O) \subseteq K$, it is clear that $\Delta \vdash_T \tilde{h}(O)$ where Δ is the set of all place Skolem constants X such that $X \in K$.

We say that h is a *solution* of $\mathcal{Q}$ if

$$\begin{array}{rcrcl}
u \leq_O v \text{ in } \mathcal{Q} & : & \tilde{h}(u) & \leq & \tilde{h}(v) \\
O \subseteq_O K \text{ in } \mathcal{Q} & : & \tilde{h}(O) & \subseteq & K \\
H_a \leq_H H_b \text{ in } \mathcal{Q} & : & \tilde{h}(H_a) & \leq & \tilde{h}(H_b) \\
H_a \in_H K \text{ in } \mathcal{Q} & : & \tilde{h}(H_a) & \in & K \\
H_a \neq_H \mathsf{unkn} \text{ in } \mathcal{Q} & : & \tilde{h}(H_a) & \neq & \mathsf{unkn}
\end{array}$$

Constraint generation. We now show how to map a term to a constraint system. For a term a, we define an ACD-system $(\mathcal{V}_a, \mathcal{W}_a, \mathcal{Q}_a)$. The set $\mathcal{V}_a$ consists of a variable O_c for each occurrence of a subterm c of a, a variable $O_{\overline{c.l_j}}$ for each occurrence of a subterm $c.l_j$ of a and a variable $O^\bullet_x$ for each bound variable x. The set $\mathcal{W}_a$ consists of two variables H_c, H'_c for each occurrence of a subterm c of a, variable $H_{\overline{c.l_j}}$ for each occurrence of a subterm $c.l_j$ of a and a variable $H^\bullet_x$ for each bound variable x. Intuitively, O_c and H_c denote respectively the type of the object part and place type of c "after" subtyping, $O_{\overline{c.l_j}}$ and $H_{\overline{c.l_j}}$ denote the object part and place type of $c.l_j$ "before" subtyping and H'_c is the place type of c's place of evaluation. Additionally, $O^\bullet_x$ and $H^\bullet_x$ denote the two parts of the type that one could have declared for x. We use the rules in Figure 3 to generate the set $\mathcal{Q}_a$. Specifically, we use judgments of the form $\Delta; \bar{\Gamma} \vdash a : \mathcal{Q}_a$ to denote that for a term a in the context $(\Delta; \bar{\Gamma})$, we derive the constraint set $\mathcal{Q}_a$. Here, $\bar{\Gamma}$ is the domain of Γ. For simplicity, we use $u =_O v$ to denote the two constraints $u \leq_O v$ and $v \leq_O u$. Similarly, $u =_H v$ denotes the two constraints $u \leq_H v$ and $v \leq_H u$.

Given a constraint solution h and an environment Γ, we say that h *extends* Γ, written $h \rhd \Gamma$, if and only if $\forall x \in \mathit{dom}(\Gamma) : \Gamma(x) = (h(O_x), h(H_x))$.

Theorem 3 shows that we can think of typability of a term c in terms of satisfiability of $\mathcal{Q}_c$.

▶ **Theorem 3** (From Types to Constraints).
$\vdash_P (\Delta, \Gamma, \pi_c, c, C)$ *if and only if* $\Delta; \bar{\Gamma} \vdash c : \mathcal{Q}_c$ *and there exists a solution* h *for*

$$\tilde{\mathcal{Q}}_c = \mathcal{Q}_c \ \cup \ \left\{ \bigcup_{y \in \bar{\Gamma}} (O_y \subseteq_O \mathcal{D} \cup \mathsf{unkn}, H_y \in_H \mathcal{D} \cup \mathsf{unkn}) \right\} \cup \{H'_c \in_H \mathcal{D}\}$$

(where $\mathcal{D} \equiv \Delta \cup \mathsf{Places}$*) such that*

$$h \rhd \Gamma \ \wedge \ \bar{\Gamma} = \mathit{dom}(\Gamma) \ \wedge \ \tilde{h}(H'_c) = \pi_c \ \wedge \ (\tilde{h}(O_c), \tilde{h}(H_c)) = C \ .$$

$$\text{(C-Var)} \quad \frac{x \in \bar{\Gamma}}{\Delta; \bar{\Gamma} \ \vdash \ x : \{O_x \leq_O O^{\bullet}_x, H_x \leq_H H^{\bullet}_x\}}$$

$$\text{(C-Obj)} \quad \frac{\begin{array}{l} \Delta; (\bar{\Gamma}, x_j) \ \vdash \ b_j : \mathcal{Q}_{b_j} \qquad \forall j \in 1..n \\ o \equiv [l_i = \varsigma(x_i) \ b_i \ ^{i \in 1..n}] \\ \mathcal{Q} = \mathcal{Q}_{b_1} \cup \mathcal{Q}_{b_2} \cup ... \cup \mathcal{Q}_{b_n} \cup \\ \{ \ [l_i : (O_{b_i}, H_{b_i}) \ ^{i \in 1..n}] \leq_O O_o, H'_o \leq_H H_o, \\ \quad \forall j \in 1..n : O_{x_j} =_O [l_i : (O_{b_i}, H_{b_i}) \ ^{i \in 1..n}], \\ \quad O_{x_j} \subseteq_O \Delta \cup \mathsf{Places} \cup \{\mathsf{unkn}\}, \ H'_o =_H H_{x_j}, \ H'_o =_H H'_{b_j} \} \end{array}}{\Delta; \bar{\Gamma} \ \vdash \ [l_i = \varsigma(x_i) \ b_i \ ^{i \in 1..n}] : \mathcal{Q}}$$

$$\text{(C-Call)} \quad \frac{\begin{array}{l} \Delta; \bar{\Gamma} \ \vdash \ a : \mathcal{Q}_a \qquad \mathcal{Q} = \mathcal{Q}_a \cup \{ \ O_a \leq_O [l_j : (\bar{O}_{a.l_j}, \bar{H}_{a.l_j})], \\ \quad \bar{O}_{a.l_j} \leq_O O_{a.l_j}, \ \bar{H}_{a.l_j} \leq_H H_{a.l_j}, \ H_a =_H H'_a, \ H'_{a.l_j} =_H H'_a\} \end{array}}{\Delta; \bar{\Gamma} \ \vdash \ a.l_j : \mathcal{Q}}$$

$$\text{(C-Update)} \quad \frac{\begin{array}{l} \Delta; \bar{\Gamma} \ \vdash \ a : \mathcal{Q}_a \qquad \Delta; (\bar{\Gamma}, x) \ \vdash \ b : \mathcal{Q}_b \qquad o \equiv a.l_j \Leftarrow \varsigma(x) \ b \\ \mathcal{Q} = \mathcal{Q}_a \cup \mathcal{Q}_b \cup \\ \{ \ O_a \leq_O O_o, \ H_a \leq_H H_o, \ O_a \leq_O [l_j : (O_b, H_b)], \ O_a =_O O_x, \ H_a =_H H_x, \\ \quad H_a =_H H'_a, \ H'_o =_H H'_a, \ H'_o =_H H'_b \ \ \} \end{array}}{\Delta; \bar{\Gamma} \ \vdash \ a.l_j \Leftarrow \varsigma(x) \ b : \mathcal{Q}}$$

$$\text{(C-AtObject)} \quad \frac{\begin{array}{l} \Delta; \bar{\Gamma} \ \vdash \ a : \mathcal{Q}_a \qquad \Delta; \bar{\Gamma} \ \vdash \ b : \mathcal{Q}_b \qquad o \equiv at(a.place) \ b \\ \mathcal{Q} = \mathcal{Q}_a \cup \mathcal{Q}_b \cup \\ \{ \ O_b \leq_O O_o, \ H_b \leq_H H_o, \ H'_a =_H H'_o, \ H'_b =_H H_a, \ H_a \in_H \Delta \cup \mathsf{Places} \ \} \end{array}}{\Delta; \bar{\Gamma} \ \vdash \ at(a.place) \ b : \mathcal{Q}}$$

$$\text{(C-AtConst)} \quad \frac{\begin{array}{l} \Delta; \bar{\Gamma} \ \vdash \ b : \mathcal{Q}_b \\ \mathcal{Q} = \mathcal{Q}_b \cup \{ \ O_b \leq_O O_{at(\rho) \ b}, \ H_b \leq_H H_{at(\rho) \ b}, \ H'_b \in_H \{\rho\} \ \} \end{array}}{\Delta; \bar{\Gamma} \ \vdash \ at(\rho) \ b : \mathcal{Q}}$$

$$\text{(C-Open)} \quad \frac{\begin{array}{l} \Delta; \bar{\Gamma} \ \vdash \ a : \mathcal{Q}_a \qquad (\Delta, X); (\bar{\Gamma}, x) \ \vdash \ b : \mathcal{Q}_b \qquad o \equiv open \ x = a \ in \ b \qquad (X \notin \Delta) \\ \mathcal{Q} = \mathcal{Q}_a \cup \mathcal{Q}_b \cup \\ \{ \ O_b \leq_O O_o, \ H_b \leq_H H_o, \ O_a =_O O_x, \ H_x \in_H \{X\}, \ H'_o =_H H'_a, \\ \quad H'_o =_H H'_b, \ H_b \in_H \Delta \cup \mathsf{Places} \cup \{\mathsf{unkn}\}, \ O_b \subseteq_O \Delta \cup \mathsf{Places} \cup \{\mathsf{unkn}\} \ \} \end{array}}{\Delta; \bar{\Gamma} \ \vdash \ open \ x = a \ in \ b : \mathcal{Q}}$$

Figure 3 Constraint generation rules.

We prove Theorem 3 in Appendix C of the full version of the paper [9]. Notice that the size of the generated constraint system is linear in the size of the program. In the following section we show how to decide in polynomial time whether an ACD-system, such as $\tilde{\mathcal{Q}}_c$, has a solution.

5 Type Inference

We first define three central notions that we use to solve ACD-systems: closure, consistency, and well-formedness. Then we state our algorithm, analyze its complexity, and give an example of how it works.

$$\text{(Closure-}\leq_O\text{-1)} \quad \frac{u \leq_O v \qquad v \leq_O w}{u \leq_O w}$$

$$\text{(Closure-}\leq_O\text{-2)} \quad \frac{\begin{array}{c} u \leq_O [l_i : (O_{b_i}, H_{b_i})\ ^{i \in 1..n}] \\ u \leq_O [l'_i : (O'_{b_i}, H'_{b_i})\ ^{i \in 1..m}] \end{array}}{\forall l_i = l'_i,\ O_{b_i} =_O O'_{b_i} \qquad H_{b_i} =_H H'_{b_i}}$$

$$\text{(Closure-}\subseteq_O\text{)} \quad \frac{O_a \leq_O [l_i : (O_{b_i}, H_{b_i})\ ^{i \in 1..n}] \qquad O_a \subseteq_O K}{\forall O_{b_i}, O_{b_i} \subseteq_O K \qquad \forall H_{b_i}, H_{b_i} \in_H K}$$

$$\text{(Closure-}\leq_H\text{)} \quad \frac{H_a \leq_H H_b \qquad H_b \leq_H H_c}{H_a \leq_H H_c}$$

$$\text{(Closure-}\neq_H\text{)} \quad \frac{H_a \in_H K \qquad \mathsf{unkn} \notin K}{H_a \neq_H \mathsf{unkn}} \qquad \text{(Closure-}\in_H\text{-1)} \quad \frac{H_a \leq_H H_b \qquad H_a \in_H K}{H_b \in_H K \cup \{\mathsf{unkn}\}}$$

$$\text{(Closure-}\in_H\text{-2)} \quad \frac{H_a \leq_H H_b \qquad H_b \in_H K \qquad H_b \neq_H \mathsf{unkn}}{H_a \in_H K}$$

Figure 4 Constraint Closure Rules.

Theorem 5 shows how closure, consistency, and well-formedness together characterize the solvability of ACD-systems. Intuitively, the closure process makes all useful facts explicit, and if none of those facts contradict well-formedness or consistency, then the constraint system is satisfiable.

5.1 Closure, consistency, and well-formedness

We use N to range over the constraint elements of the form $[l_i : (O_i, H_i)\ ^{i \in 1..n}]$.

The *closure* of an ACD-system $\mathcal{Q}$ is the union of $\mathcal{Q}$ and the constraints that can be proved from constraints in $\mathcal{Q}$ using the rules in Figure 4. In some cases, a collection of constraints may enable multiple rules to be applied; the closure contains all conclusions that can be proved. The first two rules enable straightforward reasoning about constraints on object types, while the last four rules enable straightforward reasoning about constraints on place types. The remaining Rule (Closure-$\subseteq_O$) addresses the challenge stated in Section 1, namely the constraints of the form $O \subseteq_O K$. As stated in Section 1, the challenge is that a solution to $O \subseteq_O K$ may assign O a deeply nested type and we need to know that every level uses only places in K. Rule (Closure-$\subseteq_O$) brings possible problems to the top level by (1) propagating constraints of the form $O \subseteq_O K$ "downward" when possible and (2) generating place constraints along the way. This approach to connect reasoning about constraints on object types and constraints on place types is novel and powerful.

▶ **Theorem 4** (Closure Preserves Solutions). *An ACD-system and its closure have the same set of solutions.*

We prove Theorem 4 in Appendix D of the full version of the paper [9].

We say that an ACD-system $(\mathcal{V}, \mathcal{W}, \mathcal{Q})$ is *well-formed* if and only if for every constraint in $\mathcal{Q}$ of form $s \leq_O u$, where $s \equiv [\ldots]$ and $u \equiv [l : \ldots, \ldots]$, then $s \equiv [l : \ldots, \ldots]$. Intuitively,

we require that if two record types are related by $\leq_O$ and the right-hand side has an l field, then the left-hand side does as well. We have borrowed this notion of well-formedness from Palsberg's paper [20].

We say that $\mathcal{Q}$ is *consistent* if and only if for any $s \in \mathcal{W}$ and constraints in $\mathcal{Q}$ of the form $s \in_H K_1, \ldots, s \in_H K_n$, we have $(\bigcap_{i=1}^{n} K_i) \neq \emptyset$.

Intuitively, consistency ensures that we can solve all those n constraints. This notion of consistency is novel, and while it is simple, it is just what we need to complete our characterization of satisfiability (Theorem 5).

In Appendix E of the full version of the paper [9], we show how to map an ACD-system to an automaton $\mathcal{M}_s$ that represents a type $t_{\mathcal{M}_s}$ for a type variable $s \in (\mathcal{V} \cup \mathcal{W})$. The construction of the automaton is an extension of the construction in Palsberg's paper [20]. The following section shows an example of such an automaton. The automaton may have a cycle with at least one non-ϵ transition, in which case the automaton represents a *recursive* type, rather than a finite type as defined in Section 2. Our algorithm in Section 5.2 checks whether the automaton has such a cycle. We will use the notation $\lambda s : (\mathcal{V} \cup N \cup \mathcal{W}).t_{\mathcal{M}_s}$ to denote a function that has a single argument s drawn from the set $(\mathcal{V} \cup N \cup \mathcal{W})$, and which returns $t_{\mathcal{M}_s}$.

▶ **Theorem 5** (Solvability Characterization)**.** *A closed ACD-system* $(\mathcal{V}, \mathcal{W}, \mathcal{Q})$ *is solvable with recursive types if and only if it is well-formed and consistent. If it is solvable, then* $\lambda s : (\mathcal{V} \cup N \cup \mathcal{W}).t_{\mathcal{M}_s}$ *is a solution.*

Theorem 5 shows that for a closed ACD-system, we can check for satisfiability with recursive types by checking well-formedness and consistency. We prove Theorem 5 in Appendix E of the full version of the paper [9].

5.2 Type inference algorithm

Give a term a, we solve the type inference problem (stated in Section 2) with the following five-steps algorithm:

1. Use $\emptyset; \emptyset \vdash a : \mathcal{Q}_a$ (Figure 3) to generate $\mathcal{Q}_a$.
2. Map $\mathcal{Q}_a$ to $\tilde{\mathcal{Q}_a}$ (as defined in Theorem 3).
3. Close $\tilde{\mathcal{Q}_a}$ (Figure 4).
4. Check whether the closure of $\tilde{\mathcal{Q}_a}$ is well formed and consistent, and whether in the automata $t_{\mathcal{M}_{O_a}}$ and $t_{\mathcal{M}_{H_a}}$, every cycle has only ϵ-transitions.
5. If the checks in Step 4 all return Yes, then output $(t_{\mathcal{M}_{O_a}}, t_{\mathcal{M}_{H_a}})$ as the type of a, and otherwise output that a fails to type check.

Implementation. We have implemented our algorithm and run it on the programs in Section 3 and also some additional programs. In every case our implementation produced the expected result.

Correctness. Our algorithm is correct because (1) from Theorem 3 we have that the type inference problem is solvable if and only if the constraint set $\mathcal{Q}$ is solvable; (2) from Theorem 4 we have that $\mathcal{Q}$ is solvable if and only if the closure is solvable; (3) from Theorem 5 we have that the closure is solvable with recursive types if and only if the closure is well-formed and consistent; and (4) an automaton can represent an infinite type only via a cycle with at least one non-ϵ-transition.

5.3 Time complexity

The following observation is helpful to establish a lower bound on our type inference problem. Our calculus is a *conservative extension* of the Abadi-Cardelli calculus with finite first-order types and no subtyping [1]. To see this, let $\bar{a}$ range over terms in the Abadi-Cardelli calculus, that is, terms built from just variables, objects, method call, and method update. Additionally, let $\bar{A}, \bar{B}$ range over types of the form $[l_i : \bar{B}_i{}^{i \in 1..n}]$, and let $\bar{\Gamma}$ range over type environments that map variable names to types of the form $[l_i : \bar{B}_i{}^{i \in 1..n}]$. Finally, let judgments of the form $\bar{\Gamma} \vdash_{AC} \bar{a} : \bar{A}$ be those of the Abadi-Cardelli calculus with finite first-order types and no subtyping [1]. As a helper function, define *ext* to denote the function that maps a type environment $\bar{\Gamma}$ and a place type π_c to a type environment that maps a name x in the domain of $\bar{\Gamma}$ to $(\bar{\Gamma}(x), \pi_c)$.

▶ **Theorem 6** (Conservative extension). $\bar{\Gamma} \vdash_{AC} \bar{a} : \bar{A}$ *if and only if* $\Delta, ext(\bar{\Gamma}, \pi_c), \pi_c \vdash \bar{a} : (\bar{A}, \pi_c)$.

The proof of Theorem 6 is straightforward: in each direction, do induction on the structure of the type derivation. Intuitively, the proof goes through easily because Δ and π_c don't change for terms in Abadi-Cardelli calculus.

We are now ready to state the complexity of the type inference problem.

▶ **Theorem 7** (Complexity of Type Inference). *The type inference problem is P-complete and solvable in $O(n^3)$ time, where n is the size of the program.*

Proof. Let us first consider the lower bound. Palsberg [20] showed that type inference for the Abadi-Cardelli calculus with finite first-order types and no subtyping is P-hard. We can combine that result with conservative extension (Theorem 6) and get that type inference for our calculus is P-hard.

Let us then consider the upper bound. We need to consider the execution times of closure, check for well-formedness, check for consistency, and cycle detection. We will show that we can solve each of those problems in no worse than $O(n^3)$ time.

We can bound the time to compute the closure of a constraint system by application of McAllester's meta-complexity theorem [17, Theorem 1]. McAllester's theorem says that if we have a set of closure rules R, then we can map a constraint system D' to a closure D' of D in time $O(|D'| + |P_R(D')|)$, where $P_R(D')$ is the set of all *prefix firings* in D' of rules in R. We won't recall the definition of prefix firings here but merely note that for our closure rules it is straightforward to show that $|P_R(D')| = O(n^3)$. Additionally, it is straightforward to show that $|D'| = O(n^2)$. In summary, $O(|D'| + |P_R(D')|) = O(n^3)$.

We can check well-formedness with a small variation of Palsberg's algorithm [20] to check well-formedness of constraints for type inference for the Abadi-Cardelli calculus. Palsberg's algorithm runs in $O(n^2)$ time.

We can check consistency by computing $O(n)$ intersections and unions of $O(n)$ sets of $O(n)$ elements. We represent each set as a bit vector and do the check in $O(n^3)$ time.

We can check for cycles with at least one non-ϵ-transition in $O(n^2)$ time. ◀

5.4 Example

We now demonstrate our technique on a variant of Example 9 from Section 3. Consider the following program,

$$o \;\equiv\; open\ x = at(1)\ [l = \varsigma(y)\ []]\ in\ at(x.place)\ x.l\ .$$

$$o \quad \left[\begin{array}{ll} O_b \leq_O O_o & H_b \leq_H H_o \\ O_x =_O O_a & H_x \in_H \{X\} \\ O_b \subseteq_O \{0, 1, \mathsf{unkn}\} & H_b \in_H \{0, 1, \mathsf{unkn}\} \\ & H'_o =_H H'_a \\ & H'_o =_H H'_b \end{array} \right.$$

$$a \quad \left[\begin{array}{ll} O_c \leq_O O_a & H_c \leq_H H_a \\ & H'_c \in_H \{1\} \end{array} \right.$$

$$c \quad \left[\begin{array}{ll} [l : (O_{[]}, H_{[]})] \leq_O O_c & H'_c \leq_H H_c \\ O_y =_O [l : (O_{[]}, H_{[]})] & H'_c =_H H_y \\ O_y \subseteq_O \{0, 1, \mathsf{unkn}\} & H'_c =_H H'_{[]} \end{array} \right.$$

$$[] \quad \left[\begin{array}{ll} [\,] \leq_O O_{[]} & H'_{[]} \leq_H H_{[]} \end{array} \right.$$

$$b \quad \left[\begin{array}{ll} O_{x.l} \leq_O O_b & H_{x.l} \leq_H H_b \\ & H'_{x_1} =_H H'_b \\ & H'_{x.l} =_H H_{x_1} \\ & H_{x_1} \in_H \{0, 1, X\} \end{array} \right.$$

$$x.l \quad \left[\begin{array}{ll} O_{x_2} \leq_O [l : (O_{\bar{x}.l}, H_{\bar{x}.l})] & H_{\bar{x}.l} \leq_H H_{x.l} \\ O_{\bar{x}.l} \leq_O O_{x.l} & H_{x_2} =_H H'_{x_2} \\ & H'_{x_2} =_H H'_{x.l} \end{array} \right.$$

$$x_1 \quad \left[\begin{array}{ll} O_x \leq_O O_{x_1} & H_x \leq_H H_{x_1} \end{array} \right.$$

$$x_2 \quad \left[\begin{array}{ll} O_x \leq_O O_{x_2} & H_x \leq_H H_{x_2} \end{array} \right.$$

Figure 5 Constraints for the example program.

Figure 5 shows the generated constraint set for o and its subterms, assuming we start at place 0 and run on two places 0 and 1. We use the abbreviations $a \equiv at(1)\ [l = \varsigma(y)\ []], c \equiv [l = \varsigma(y)\ []], b \equiv at(x.place)\ x.l$. Variables generated for the two instances of x are shown with the appropriate subscript.

We get the following closed set of constraints:

- $[] \leq_O O_{[]} \leq_O O_{\bar{x}.l} \leq_O O_{x.l} \leq_O O_b \leq_O O_o$
- $O_y \leq_O [l : (O_{[]}, H_{[]})] \leq_O O_c \leq_O O_a \leq_O O_x \leq_O O_{x_1}$
- $O_y \leq_O [l : (O_{[]}, H_{[]})] \leq_O O_c \leq_O O_a \leq_O O_x \leq_O O_{x_2} \leq_O [l : (O_{x.l}, H_{x.l})]$
- $O_{\bar{x}.l} =_O O_{[]}, O_x =_O O_a, O_y =_O [l : (O_{[]}, H_{[]})]$
- $H_x \leq_H H_{x_2} \leq_H H'_{x_2} \leq_H H'_{x.l} \leq_H H_{x_1}$
- $H_y \leq_H H'_c \leq_H H'_{[]} \leq_H H_{[]} \leq_H H_{\bar{x}.l} \leq_H H_{x.l} \leq_H H_b \leq_H H_o$
- $H'_{[]} \leq_H H'_c \leq_H H_c \leq_H H_a$
- $H_y \leq_H H'_c \leq_H H_c \leq_H H_a$
- $H_{[]} =_H H_{\bar{x}.l}, H'_o =_H H'_a =_H H'_b =_H H'_{x_1}, H_{x_2} =_H H'_{x_2} =_H H'_{x.l} =_H H_{x_1}, H_y =_H H'_c =_H H'_{[]}$
- $H_y, H'_c, H'_{[]} \in_H \{1\}, \in_H \{1, \mathsf{unkn}\}$
- $H_x, H_{x_2}, H'_{x_2}, H'_{x.l}, H_{x_1} \in_H \{X\}, \in_H \{X, \mathsf{unkn}\}, \in_H \{0, 1, X\}$
- $H'_o, H'_a, H'_b, H'_{x_1} \in_H \{0\}, \in_H \{0, \mathsf{unkn}\}$
- $H_{[]}, H_{\bar{x}.l}, H_{x.l}, H_b, H_o \in_H \{0, 1, \mathsf{unkn}\}$
- $H_a, H_c \in_H \{1, \mathsf{unkn}\}$
- $O_y, O_{[]}, O_{\bar{x}.l}, O_{x.l}, O_b, O_o \subseteq_O \{0, 1, \mathsf{unkn}\}$

It is straightforward to check that the constraint set is well-formed and consistent, hence solvable. The solution automaton is shown in Figure 6.

For a detailed understanding of the automaton, please consult the appendices of the full version of the paper [9]. Here we merely note that for readability, we omit redundant and self ϵ-transitions from the figure. Also for each node, a so-called *labeling function value* is shown in superscript. Figure 7 shows the final type derivation for o. Note that the place check succeeds for the assigned types. Also note that since a is assigned a packed type, the program will still type check if the body of a is changed to evaluate to an object at a place other than 1.

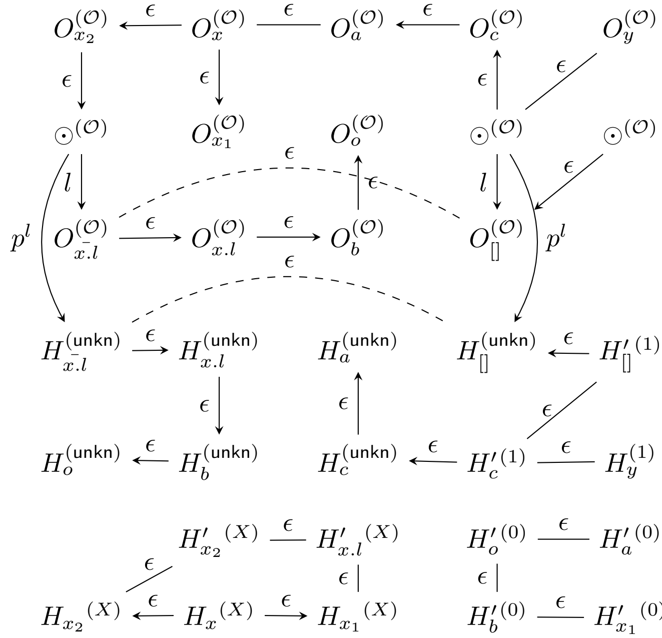


Figure 6 Automaton for the example program.

6 Discussion

In addition to the algorithm for type inference with finite types in Section 5.2, we can also do type inference with recursive types, simply by omitting the test for finiteness in Step 4 of the algorithm.

We believe that our type system is sound both for the "functional" semantics of method update in Section 2 (Theorem 1) and also for an "imperative" semantics.

For simplicity, places are not values in our calculus, yet we see no major obstacle to work with places as values.

Our algorithm can be modified to apply to the variant of Featherweight X10 that we used in the example in Section 1, as we will explain now. The only step that needs modification is constraint generation; everything else stays unchanged. The needed modifiations to the constraint generation rules in Figure 3 are rather modest. First, the rules for variables, calls, and *at* can stay essentially unchanged. Second, the rule for objects must be modified to work instead for classes and the `new` expression. Third, the rule for update must be modified to work instead for assignment and parameter passing. Fourth, the rule for open must be modified to work instead for `final`.

7 Related Work

Our work builds and improves on previous work on type systems and type inference for distributed programs. Most of the previous work comes in two flavors: object oriented or based on π-calculus. While the underlying languages are rather different, the challenges for type inference are rather similar. For object-oriented languages the challenge is about local access to fields and methods, while for π-calculus the challenge is about local access to channels.

Figure 8 gives a ten-dimensional comparison of our paper and three previous papers. The first two papers by Sewell [24] and Lhoussaine [15] are based on π-calculus [18], while the

$$\cfrac{\cfrac{\cfrac{\emptyset; y : (T_C, 1); 1 \;\vdash\; [\,] : ([], 1) \qquad ([], 1) \leq \mathit{packed}\ []}{\emptyset; y : (T_C, 1); 1 \;\vdash\; [\,] : \mathit{packed}\ []}}{\begin{array}{c}\emptyset; \emptyset; 1 \;\vdash\; c : (T_C, 1) \\ (T_C, 1) \leq \mathit{packed}\ T_C\end{array}}}{\cfrac{\emptyset; \emptyset; 1 \;\vdash\; c : \mathit{packed}\ T_C}{\emptyset; \emptyset; 0 \;\vdash\; a : \mathit{packed}\ T_C}}$$

$$\cfrac{X; x : (T_C, X); 0 \;\vdash\; x_1 : (T_C, X) \qquad \cfrac{\begin{array}{c}X; x : (T_C, X); X \;\vdash\; x_2 : (T_C, X) \\ \boldsymbol{X = X}\end{array}}{X; x : (T_C, X); X \;\vdash\; x.l : \mathit{packed}\ []}}{X; x : (T_C, X); 0 \;\vdash\; b : \mathit{packed}\ []}$$

$$\cfrac{\emptyset; \emptyset; 0 \;\vdash\; a : \mathit{packed}\ T_C \qquad X; x : (T_C, X); 0 \;\vdash\; b : \mathit{packed}\ []}{\emptyset; \emptyset; 0 \;\vdash\; o : \mathit{packed}\ []}$$

Figure 7 Type derivation for the example program. Abbreviation: $T_C \equiv [l : \mathit{packed}\ []]$.

	Sewell [24]	Lhoussaine [15]	Chandra et al. [5]	Haque & Palsberg [this paper]
Objects			√	√
Places	√	√	√	√
Place checks	√	√	√	√
at(constant place)	√	√	√	√
$at(x.place)$			√	√
Existential types		√		√
Place-oblivious objects				√
Subtyping	√	√	√	√
Type inference	√	√	√	√
Type inference in P-time				√

Figure 8 Comparison.

paper by Chandra et al. [5] and our paper are based on object-oriented languages. Each of the four papers supports places and places checks, and has a construct for place shift to a constant place, such as $at(\rho)b$ in our paper. The four papers agree on the semantics of place shift to a constant place.

The papers by Sewell [24] and Lhoussaine [15] have no notion of a place shift to a statically unknown place, such as $at(x.place)$ in our paper. Thus, one cannot express place-oblivious objects in their calculi. In contrast, the paper by Chandra et al. [5] and our paper both have notions of $at(x.place)$. One of the goals of the type system of Chandra et al. [5] is to track place correlation between objects. In particular, they provide an approach to determine whether two fields or expressions have the *same* place type. This enables successful type checking of several common programming patterns, yet leaves other open problems. In particular, the unification-based approach of Chandra et al. fails on place-oblivious objects. For example if a field f at one time references an object at place 1 and at another time references an object at place 2, and the two places 1 and 2 don't unify.

The papers by Sewell [24] and Lhoussaine [15] are part of a long line of π-calculus-based work on location-aware computation that includes [11, 4, 3]. The calculi in those papers

provide explicit language constructs for *locations* (which we call places) and *agent migration*. Notably, Hennessy and Riely [11] present an extension to the π-calculus with light-weight existential types for ensuring locality of channel communication in well-typed programs. Amadio et al. [4, 3] prove local deadlock-freedom (*receptiveness*) for a simplified version of the Hennessy-Riely calculus. Lhoussaine [15] presents an inference algorithm for a simplified version of the Hennessy-Riely calculus.

The approach of Hennessy and Riely [11] differs from ours in significant ways. In particular, the Hennessy-Riely calculus treats locations as first-class values, and a programmer has to specify explicitly the dependence between a channel name r and a location l. The resulting compound term, denoted $r@l$, serves as an explicit constructor and destructor for an existential type. In contrast, our calculus is more concise in its treatment of existentials because we use implicit subtyping to introduce a form of existential type.

The calculus of Hennessy and Riely [11] can encode a place shift. An expression such as $at(1)\ at(x.place)\ y.f$, which is place safe if x and y are at the same place, can in the Hennessy-Riely calculus be emulated by:

$$1[\![u?((x,y)@d)go\ d.y!\langle f\rangle 0]\!]\ .$$

The programmer has to use $(x, y)@d$ to specify that x and y are at the same location d. In general, the programmer has to specify any place correlation between two channels. Such place correlation can be lost if channels are quantified separately. Our calculus uses place types to track such place correlation.

The calculi in the papers [11, 4, 3, 15] use existential types in a way that is substantially different from ours. In particular, in these calculi a term can have the type

$$\exists r.\ l : loc\{r : B\}$$

our calculus quantifies the object location rather than its reference.

All four papers in Figure 8 support subtyping, though of somewhat different flavors. Sewell [24] specifies a subtyping order on channel capabilities that can distinguish local or global capabilities. Lhoussaine [15] specifies a "width" subtyping order in which a subtype defines at least the same channel names as any supertype. Finally, Chandra et al. [5] specifies a constraint-based subtype order that relates constrained types based on entailment of constraint sets. In contrast to our calculus, the calculi in the papers by Sewell [24] and Lhoussaine [15] cannot use subtyping to mask an object's location. Such subtyping is possible in the calculus of Chandra et al. but isn't fully supported by their type inference algorithm [5, Section 5].

All four papers in Figure 8 support type inference, though with completely different algorithms:

Paper	Key technique
Sewell [24]	Least upper bound of compatible types
Lhoussaine [15]	Constraint solving by unification and rewriting
Chandra et al. [5]	Constraint solving by unification
This paper	Constraint solving by closure + consistency + wellformedness

Among those four papers, only our paper has a type inference algorithm that runs in worst-case polynomial time. Sewell and Lhoussaine's papers have no discussion or results about the complexity of their inference algorithms, while Chandra et al. states that their type inference problem is undecidable (and hence that their algorithm is incomplete).

Other related work. Liblit and Aiken presented a type system and a type inference algorithm that distinguishes between local and global data [16]. The goal of their type inference algorithm is to to minimize the number of global pointers. The paper reports on an implementation for Titanium, an object-oriented language, but doesn't consider place checks, place shift, or existential types.

The idea of a type of the form $\exists\pi.([l_i : B_i \ ^{i\in 1..n}], \pi)$ stems from Jeffrey's paper on a distributed object calculus [13]. Jeffrey's calculus is a modification of the Gordon-Hankin concurrent object calculus [8] and is semantically quite different from our calculus. Intuitively, Jeffrey's calculus is about "sending the computation", while our calculus is about "sending the data." Jeffrey's calculus has no notion of place shift. Instead, it has a notion of remote spawn that superficially looks like a place shift, but more closely resembles a remote data fetch operation. Jeffrey's paper doesn't consider type inference.

The join-calculus of Fournet and Gonthier [7] has hierarchical virtual locations together with agents that can migrate between places. Their type system, however, has no notion of place checking, and the programmer has to control object distribution explicitly.

Henglein [10] presented algorithms for type inference for the Abadi-Cardelli calculus. His algorithms are faster than $O(n^3)$. However, his setting doesn't require a consistency check. Our setting requires a consistency check that with our straightforward algorithm runs in $O(n^3)$. Hence, any improvement of our bound of $O(n^3)$ must in particular improve the algorithm for consistency check.

8 Conclusion

We have presented the first type system and inference algorithm for place-oblivious objects. We believe our algorithm can be useful for programming languages such as X10. In particular, our algorithm enables a language implementation to avoid run-time place checks for place-oblivious objects.

Our calculus of distributed objects may be of independent interest. In future work we hope to extend our type system and inference algorithm with more general notions of existential types. We also hope to extend our approach to handle distributed arrays. Finally, we hope to design a calculus that does *open* implicitly.

Acknowledgments. We thank Mohsen Lesani for a meticulous review of the soundness proof, and we thank John Bender, Matt Brown, Nikolay Laptev, and the ECOOP reviewers for numerous helpful suggestions.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. Shivali Agarwal, RajKishore Barik, V. Nandivada, Rudrapatna Shyamasundar, and Pradeep Varma. Static detection of place locality and elimination of runtime checks. In G. Ramalingam, editor, *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 53–74. Springer Berlin / Heidelberg, 2008.
3. Roberto M. Amadio, Gérard Boudol, and Cédric Lhoussaine. The receptive distributed π-calculus. *ACM Trans. Program. Lang. Syst.*, 25(5):549–577, September 2003.
4. Roberto M. Amadio, Gerard Boudol, Cedric Lhoussaine, and Roberto M. Amadio. The receptive distributed π-calculus, 1999.
5. Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type inference for locality analysis of distributed data structures. In *Proceedings of the 13th ACM SIGPLAN*

Symposium on Principles and practice of parallel programming, PPoPP'08, pages 11–22, New York, NY, USA, 2008. ACM.

6 Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA'05, pages 519–538, New York, NY, USA, 2005. ACM.

7 Cedric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer Berlin Heidelberg, 2002.

8 Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing, 1998.

9 Riyaz Haque and Jens Palsberg. Type inference for place-oblivious objects. Full version of a paper with the same title in ECOOP 2015, `http://www.cs.ucla.edu/~palsberg/paper/ecoop15full.pdf`.

10 Fritz Henglein. Breaking through the n^3 barrier: Faster object type inference. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, 1997.

11 Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:2002, 1998.

12 Paul N. Hilfinger, Dan Oscar Bonachea, Kaushik Datta, David Gay, Susan L. Graham, Benjamin Robert Liblit, Geoffrey Pike, Jimmy Zhigang Su, and Katherine A. Yelick. Titanium language reference manual, version 2.19. Technical Report UCB/EECS-2005-15, EECS Department, University of California, Berkeley, Nov 2005.

13 A. S. A. Jeffrey. A distributed object calculus. In *Proc. Foundations of Object Oriented Languages*, 2000.

14 Jonathan K. Lee and Jens Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of PPOPP'10, 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, January 2010.

15 Cedric Lhoussaine. Type inference for a distributed π-calculus. In Pierpaolo Degano, editor, *Programming Languages and Systems*, volume 2618 of *Lecture Notes in Computer Science*, pages 253–268. Springer Berlin Heidelberg, 2003.

16 Ben Liblit and Alexander Aiken. Type systems for distributed data structures. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL*, pages 199–213, 2000.

17 David McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.

18 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, September 1992.

19 Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, May 1999.

20 Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995. Preliminary version in Proceedings of LICS'94, Ninth Annual IEEE Symposium on Logic in Computer Science, pages 186–195, Paris, France, July 1994.

21 Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

22 Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages.* The MIT Press, Cambridge, MA, USA, 2004.

23 Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification. Technical report, IBM, January 2012.

24 Peter Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *In Proceedings of ICALP'98, LNCS 1443*, pages 695–706. Springer-Verlag, 1998.

25 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1992.

26 Katherine A. Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul N. Hilfinger, Susan L. Graham, David Gay, Phillip Colella, and Alexander Aiken. Titanium: A high-performance Java dialect. *Concurrency – Practice and Experience*, 10(11-13):825–836, 1998.

Asynchronous Liquid Separation Types

Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis

Max Planck Institute for Software Systems, Germany
{jkloos,rupak,viktor}@mpi-sws.org

Abstract

We present a refinement type system for reasoning about asynchronous programs manipulating shared mutable state. Our type system guarantees the absence of races and the preservation of user-specified invariants using a combination of two ideas: refinement types and concurrent separation logic. Our type system allows precise reasoning about programs using two ingredients. First, our types are indexed by sets of resource names and the type system tracks the effect of program execution on individual heap locations and task handles. In particular, it allows making strong updates to the types of heap locations. Second, our types track ownership of shared state across concurrently posted tasks and allow reasoning about ownership transfer between tasks using permissions. We demonstrate through several examples that these two ingredients, on top of the framework of liquid types, are powerful enough to reason about correct behavior of practical, complex, asynchronous systems manipulating shared heap resources.

We have implemented type inference for our type system and have used it to prove complex invariants of asynchronous OCaml programs. We also show how the type system detects subtle concurrency bugs in a file system implementation.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs, D.2.4 Software/Program Verification

Keywords and phrases Liquid Types, Asynchronous Parallelism, Separation Logic, Type Systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.396

1 Introduction

Asynchronous programming is a common programming idiom used to handle concurrent interactions. It is commonly used not only in low-level systems code, such as operating systems kernels and device drivers, but also in internet services, in programming models for mobile applications, in GUI event loops, and in embedded systems.

An asynchronous program breaks the logical units that comprise its functionality into atomic, non-blocking, sections called tasks. Each task performs some useful work, and then schedules further tasks to continue the work as necessary. At run time, an application-level co-operative scheduler executes these tasks in a single thread of control. Since tasks execute atomically and the scheduler is co-operative, asynchronous programs must ensure that each task runs for a bounded time. Thus, blocking operations such as I/O are programmed in an asynchronous way: an asynchronous I/O operation returns immediately whether or not the data is available, and returns a promise that is populated once the data is available. Conversely, a task can wait on a promise, and the scheduler will run such a task once the data is available. In this way, different logical units of work can make simultaneous progress.

In recent years, many programming languages provide support for asynchronous programming, either as native language constructs (e.g., in Go [8] or Rust [30]) or as libraries (e.g., libevent [18] for C, Async [20] and Lwt [33] for OCaml). These languages and libraries allow the programmer to write and compose tasks in a monadic style and a type checker

© Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 396–420

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

enforces some basic invariants about the data being passed around. However, reasoning about finer-grained invariants, especially involving shared resources, remains challenging. Furthermore, the loose coupling between different enabled handlers which may interact through shared resources gives rise to many subtle bugs in these programs.

In this paper, we focus on asynchronous programs written in OCaml, whose type system already guarantees basic memory safety, and seek to extend the guarantees that can be provided to the programmers. Specifically, we would like to be able to automatically verify basic correctness properties such as race-freedom and the preservation of user-supplied invariants. To achieve this goal, we combine two well-known techniques, *refinement types* and *concurrent separation logic*, into a type system we call *asynchronous liquid separation types* (ALS types).

Refinement types [7, 35] are good at expressing invariants that are needed to prove basic correctness properties. For example, to ensure that that array accesses never go out of bounds, one can use types such as $\{x : int \mid 0 \leq x < 7\}$. Moreover, in the setting of *liquid types* [28], many such refinement types can be inferred automatically, relieving the programmer from having to write any annotations besides the top-level specifications. However, existing refinement type systems do not support concurrency and shared state.

On the other hand, concurrent separation logic (CSL) [22] is good at reasoning about concurrency: its rules can handle strong updates in the presence of concurrency. Being an expressive logic, CSL can, in principle, express all the invariants expressible via refinement types, but in doing so, gives up on automation. Existing fully automated separation logic tools rely heavily on shape analysis (e.g. [5, 3]) and can find invariants describing the pointer layout of data structures on the heap, but not arbitrary properties of their content.

Our combination of the two techniques inherits the benefits of each. In addition, using liquid types, we automate the search for refinement type annotations over a set of user-supplied predicates using an SMT solver. Given a program and a set of predicates, our implementation can automatically infer rich data specifications in terms of these predicates for asynchronous OCaml programs, and can prove the preservation of user-supplied invariants, as well as the absence of memory errors, such as array out of bounds accesses, and concurrency errors, such as data races. This is achieved by extending the type inference procedure of liquid types, adding a step that derives the structure of the program's heap and information about ownership of resources using an approach based on abstract interpretation. Specifically, our system was able to infer a complex invariant in a parallel SAT solve and detect a subtle concurrency bug in a file system implementation.

Because of limited space, we have omitted proofs and technical details in this version of the paper. A full version can be found on our website [14].

Outline. The remainder of our paper is structured as follows:

Section 2 We introduce a small ML-like language with asynchronous tasks, and present a number of small examples motivating the main features of our type system.

Section 3 We give a formal presentation of our type system and state the type safety theorem.

Section 4 We discuss how we perform type inference by extending the liquid typing algorithm [28] with a static analysis.

Section 5 We evaluate our type inference implementation on a number of case studies that include a file system and a parallel SAT solver. We discuss a subtle concurrency error uncovered in an asynchronous file system implementation.

Section 6 We discuss limitations of ALS types.

Section 7 We discuss related work.

$$\begin{array}{rlll} c & \text{Constants} & x, f & \text{Variables} \\ \ell \in \mathsf{Locs} & \text{Heap locations} & p \in \mathsf{Tasks} & \text{Task handles} \\ v \in \mathsf{Values} & ::= c \mid x \mid \lambda x.\, e \mid \mathbf{rec}\, f\, x\, e \mid \ell \mid p & & \\ e & ::= v \mid e\, e \mid \mathbf{ref}\ e \mid\ !e e \mid e\ \texttt{:=}\ e \mid \mathbf{post}\ e \mid \mathbf{wait}\ e \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e & & \\ t \in \mathsf{TaskStates} & ::= \mathsf{run}\colon e \mid \mathsf{done}\colon v & & \\ H & := \mathsf{Locs} \rightharpoonup \mathsf{Values} & \text{Heaps} & \\ P & := \mathsf{Tasks} \rightharpoonup \mathsf{TaskStates} & \text{Task buffers} & \end{array}$$

Figure 1 The core calculus.

$$\text{EL-POST}\ \frac{p \text{ fresh w.r.t. } P, p_r}{(\mathbf{post}\ e, H, P) \hookrightarrow_{p_r} (p, H, P[p \mapsto \mathsf{run}\colon e])} \qquad \text{EL-WAITDONE}\ \frac{P(p) = \mathsf{done}\colon v}{(\mathbf{wait}\ p, H, P) \hookrightarrow_{p_r} (v, H, P)}$$

$$\text{EG-LOCAL}\ \frac{(e, H, P) \hookrightarrow_{p_r} (e', H', P') \qquad p_r \notin \operatorname{dom} P}{(H, P[p_r \mapsto \mathsf{run}\colon e], p_r) \hookrightarrow (H', P'[p_r \mapsto \mathsf{run}\colon e'], p_r)}$$

$$\text{EG-WAITRUN}\ \frac{P(p_2) = \mathsf{run}\colon_ \quad P(p_1) = \mathsf{run}\colon \mathcal{C}[\mathbf{wait}\ p] \qquad P(p) = \mathsf{run}\colon_}{(H, P, p_1) \hookrightarrow (H, P, p_2)} \qquad \text{EG-FINISHED}\ \frac{P(p_2) = \mathsf{run}\colon_ \quad P(p_1) = \mathsf{run}\colon v \qquad p_1 \neq p_2}{(H, P, p_1) \hookrightarrow (H, P[p_1 \mapsto \mathsf{done}\colon v], p_2)}$$

Figure 2 Small-step semantics.

2 Examples and Overview

2.1 A core calculus for asynchronous programming

For concreteness, we base our formal development on a small λ calculus with recursive functions, ML-style references, and two new primitives for asynchronous concurrency: **post** e that creates a new task that evaluates the expression e and returns a handle to that task; and **wait** e that evaluates e to get a task handle p, waits for the completion of task with handle p, and returns the value that the task yields. Figure 1 shows the core syntax of the language; for readability in examples, we use standard syntactic sugar (e.g., let).

The semantics of the core calculus is largely standard, and is presented in a small-step operational fashion. We have two judgments: (1) the *local semantics*, $(e, H, P) \hookrightarrow_{p_r} (e', H', P')$, that describe the evaluation of the active task, p_r, and (2) the *global semantics*, $(H, P, p) \hookrightarrow (H', P', p')$, that describe the evaluation of the system as a whole. Figure 2 shows the local semantics rules for posts and waits, as well as the global semantic rules.

In more detail, local configurations consist of the expression being evaluated, e, the heap, H, and the task buffer, P. We model heaps as partial maps from locations to values, and task buffers as partial maps from task handles to task states. A task state can be either a running task containing the expression yet to be evaluated, or a finished task containing some value. We assume that the current process being evaluated is not in the task buffer, $p_r \notin \operatorname{dom} P$. Evaluation of a **post** e expression generates a new task with the expression e, while **wait** p reduces only if the referenced task has finished, in which case its value is returned. For the standard primitives, we follow the OCaml semantics. In particular, evaluation uses

right-to-left call-by-value reduction.

Global configurations consist of the heap, H, the task buffer, P, and the currently active task, p. As the initial configuration of an expression e, we take $(\emptyset, [p_0 \mapsto \mathsf{run}\colon e], p_0)$. A global step is either a local step (EG-Local), or a scheduling step induced by the wait instruction when the task waited for is still running (EG-WaitRun) or the termination of a task (EG-Finished). In these cases, some other non-terminated task p_2 is selected as the active task.

2.2 Promise types

We now illustrate our type system using simple programs written in the core calculus. They can be implemented easily in OCaml using libraries such as Lwt [33] and Async [20]. If expression e has type α, then **post** e has type $\mathsf{promise}\,\alpha$, a promise for the value of type α that will eventually be computed. If the type of e is $\mathsf{promise}\,\alpha$, then **wait** e types as α.

As a simple example using these operations, consider the following function that copies data from an input stream `ins` to an output stream `outs`:

```
let rec copy1 ins outs =
  let buf = wait (post (read ins)) in
  let _ = wait (post (write outs buf buf)) in
  if eof ins then () else copy1 ins outs
```

where the read and write operations have the following types:

```
read: stream → buffer        write: stream → buffer → unit
```

and `eof` checks if `ins` has more data. The code above performs (potentially blocking) reads and writes asynchronously[1]. It posts a task for reading and blocks on its return, then posts a task for writing and blocks on its return, and finally, calls itself recursively if more data is to be read from the stream. By posting `read` and `write` tasks, the asynchronous style enables other tasks in the system to make progress: the system scheduler can run other tasks while `copy1` is waiting for a read or write to complete.

2.3 Refinement types

In the above program, the ML type system provides coarse-grained invariants that ensure that the data type eventually returned from `read` is the same data type passed to `write`. To verify finer-grained invariants, in a sequential setting, one can augment the type system with *refinement types* [35, 28]. For example, in a refinement type, one can write $\{\nu : \mathtt{int} \mid \nu \geq 0\}$ for refinement of the integer type that only allows non-negative values. In general, a refinement type of the form $\{\nu : \tau \mid p(\nu)\}$ is interpreted as a subtype of τ where the values v are exactly those values of τ that satisfy the predicate $p(v)$. A subtyping relation between types $\{\nu : \tau \mid \rho_1\}$ and $\{\nu : \tau \mid \rho_2\}$ can be described informally as "all values that satisfy ρ_1 must also satisfy ρ_2"; this notion is made precise in Section 3.

For purely functional asynchronous programs, the notion of type refinements carries over transparently, and allows reasoning about finer-grain invariants. For example, suppose we know that the read operation always returns buffers whose contents have odd parity. We can express this by refining the type of `read` to $\mathtt{stream} \to \mathsf{promise}\,\{\nu : \mathtt{buffer} \mid \mathrm{odd}(\nu)\}$. Dually,

[1] We assume that reading an empty input stream simply results in an empty buffer; an actual implementation will have to guard against I/O errors.

we can require that the write operation only writes buffers whose contents have odd parity by specifying the type $\texttt{stream} \to \{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\} \to \mathsf{promise}\,\mathsf{unit}$. Using the types for **post** and **wait**, it is simple to show that the code still types in the presence of refinements.

Thus, for purely functional asynchronous programs, the machinery of refinement types and SMT-based implementations such as liquid types [28] generalize transparently and provide powerful reasoning tools. The situation is more complex in the presence of shared state.

2.4 Refinements and state: strong updates

Shared state complicates refinement types even in the sequential setting. Consider the following sequential version of copy, where read and write take a heap-allocated buffer:

```
let seqcp ins outs = let b = ref empty_buffer in readb ins b; writeb outs b
```

where $\texttt{readb, writeb: stream} \to \mathsf{ref}\,\texttt{buffer} \to \mathsf{unit}$.[2] As subtyping is unsound for references (see, e.g., [23, §15.5]), it is not possible to track the precise contents of a heap cell by modifying the refinement predicate in the reference type. One symptom of this unsoundness is that there can be multiple instances of a reference to a heap cell, say $x_1, \ldots, x_n$, with types $\mathsf{ref}\,\tau_1, \ldots, \mathsf{ref}\,\tau_n$. It can be shown that all the types $\tau_1, \ldots, \tau_n$ must be essentially the same: Suppose, for example, that $\tau_1 = \texttt{int}_{=1}$ and $\tau_2 = \texttt{int}_{\geq 0}$. Suppose furthermore that x_1 and x_2 point to the same heap cell. Then, using standard typing rules, the following piece of code would type as $\texttt{int}_{=1}$: x_2 `:= 2; !`x_1. But running the program would return 2, breaking type safety. By analogy with static analysis, we call references typed like in ordinary ML *weak references*, and updates using only weak references *weak updates*. Their type only indicates which values a heap cell can possibly take over the execution of a whole program, but not its current contents.

Therefore, to track refinements over changes in the mutable state, we modify the type system to perform *strong updates* that track such changes. For this, our type system includes preconditions and postconditions that explicitly describe the global state before and after the execution of an expression. We also augment the types of references to support strong updates, giving us *strong references*.

Resource sets. To track heap cells and task handles in pre- and postconditions, we introduce *resource names* that uniquely identify each resource. At the type level, global state is described using *resource sets* that map resource names to types. Resource sets are written using a notation inspired from separation logic. For example, the resource set $\mu \mapsto \{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}$ describes a heap cell that is identified by the resource name μ and contains a value of type $\{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}$.

To connect references to resources, reference types are extended with indices ranging over resource names. For example, the reference $\mathsf{ref}_\mu\,\texttt{buffer}$ denotes a reference that points to a heap cell with resource name μ and that contains a value of type `buffer`. In general, given a reference type $\mathsf{ref}_\mu\,\tau$ and a resource set including $\mu \mapsto \tau'$, we ensure τ' is a subtype of τ. Types of the form $\mathsf{ref}_\mu\,\tau$ are called *strong references*.

Full types. Types and resource sets are tied together by using *full types* of the form $\text{И}\Xi.\,\tau\langle\eta\rangle$, where τ is a type, η is a resource set, and Ξ is a list of resource names that are considered "not yet bound." The И binder indicates that all names in Ξ must be fresh, and therefore,

[2] We write $\mathsf{ref}\,\texttt{buffer}$ instead of $\texttt{buffer ref}$ for ease of readability.

distinct from all names occurring in the environment. For example, if expression e has type $\text{И}\mu.\, \mathsf{ref}_\mu\, \tau\langle \mu \mapsto \tau' \rangle$, it means that e will return a reference to a newly-allocated memory cell with a fresh resource name μ, whose content has type τ', and $\tau' \preceq \tau$.

We use some notational shorthands to describe full types. We omit quantifiers if nothing is quantified: $\text{И}\cdot.\, \tau\langle\eta\rangle = \tau\langle\eta\rangle$ and $\forall \cdot .\tau = \tau$. If a full type has an empty resource set, it is identified with the type of the return value, like this: $\tau\langle \mathsf{emp} \rangle = \tau$.

To assign a full type to an expression, the global state in which the expression is typed must be given as part of the environment. More precisely, the typing judgment is given as $\Gamma; \eta \vdash e : \text{И}\Xi.\, \tau\langle\eta'\rangle$. It reads as follows: in the context Γ, when starting from a global state described by η, executing e will return a value of type τ and a global state matching η', after instantiating the resource names in Ξ. As an example, the expression `ref empty_buffer` would type as $\cdot; \mathsf{emp} \vdash \ldots : \text{И}\mu.\, \mathsf{ref}_\mu\, \mathtt{buffer}\langle \mu \mapsto \mathtt{buffer} \rangle$.

To type functions properly, we need to extend function types to capture the functions' effects. For example, consider an expression e that types as $\Gamma, x : \tau_x; \eta \vdash e : \varphi$. If we abstract it to a function, its type will be $x\colon \tau_x\langle\eta\rangle \to \varphi$, describing a function that takes an argument of type τ_x and, if executed in a state matching η, will return a value of full type φ.

Furthermore, function types admit name quantification. Consider the expression e given by `!x + 1`. Its type is $\mu, x : \mathsf{ref}_\mu\, \mathtt{int}; \mu \mapsto \mathtt{int} \vdash e : \mathtt{int}\langle \mu \mapsto \mathtt{int} \rangle$. By lambda abstraction,

$$\mu; \mathsf{emp} \vdash \lambda x.e : \tau\langle \mathsf{emp} \rangle \text{ with } \tau = x\colon \mathsf{ref}_\mu\, \mathtt{int}\langle \mu \mapsto \mathtt{int} \rangle \to \mathtt{int}\langle \mu \mapsto \mathtt{int} \rangle.$$

To allow using this function with arbitrary references, the name μ can be universally quantified:

$$\cdot; \mathsf{emp} \vdash \lambda x.e : \tau\langle \mathsf{emp} \rangle \text{ with } \tau = \forall \mu.(x\colon \mathsf{ref}_\mu\, \mathtt{int}\langle \mu \mapsto \mathtt{int} \rangle \to \mathtt{int}\langle \mu \mapsto \mathtt{int} \rangle).$$

In the following, if a function starts with empty resource set as a precondition, we omit writing the resource set: $x\colon \tau_x\langle \mathsf{emp} \rangle \to \varphi$ is written as $x : \tau_x \to \varphi$.

As an example, the type of the `readb` function from above would be:

$$\mathtt{readb} : \mathtt{stream} \to \forall \mu.(b\colon \mathsf{ref}_\mu\, \mathtt{buffer}\langle \mu \mapsto \mathtt{buffer} \rangle \to \mathsf{unit}\langle \mu \mapsto \{\nu : \mathtt{buffer} \mid \text{odd } \nu\} \rangle$$

$$\mathtt{writeb} : \mathtt{stream} \to$$
$$\forall \mu.(b\colon \mathsf{ref}_\mu\, \mathtt{buffer}\langle \mu \mapsto \{\nu : \mathtt{buffer} \mid \text{odd } \nu\} \rangle \to \mathsf{unit}\langle \mu \mapsto \{\nu : \mathtt{buffer} \mid \text{odd } \nu\} \rangle$$

2.5 Asynchrony and shared resources

The main issue in ensuring safe strong updates in the presence of concurrency is that aliasing control now needs to extend across task boundaries: if task 1 modifies a heap location, all other tasks with access to that location must be aware of this. Otherwise, the following race condition may be encountered: suppose task 1 and task 2 are both scheduled, and heap location ξ_1 contains the value 1. During its execution, task 1 modifies ξ_1 to hold the value 2, whereas task 2 outputs the content of ξ_1. Depending on whether task 1 or task 2 is run first by the scheduler, the output of the program differs. A precise definition of race conditions for asynchronous programs can be found in [26].

To understand the interaction between asynchronous calls and shared state, consider the more advanced implementation of the copying loop that uses two explicitly allocated buffers and a double buffering strategy:

```
let copy2 ins outs =
  let buf1 = ref empty_buffer and buf2 = ref empty_buffer in
  let loop bufr bufw =
    let drain_bufw = post (writeb outs bufw) in
    if eof ins then wait drain_bufw else
      let fill_bufr = post (readb ins bufr) in
      wait drain_bufw; wait fill_bufr; loop bufw bufr
  in wait (post (readb ins buf1));
  loop buf2 buf1
```

where `readb` : $\texttt{stream} \to \mathsf{ref}\,\texttt{buffer} \to \mathsf{unit}$ and `writeb` : $\texttt{stream} \to \mathsf{ref}\,\texttt{buffer} \to \mathsf{unit}$. The double buffered copy pre-allocates two buffers, `buf1` and `buf2` that are shared between the reader and the writer. After an initial read to fill `buf1`, the writes and reads are pipelined so that at any point, a read and a write occur concurrently.

A key invariant is that the buffer on which `writeb` operates and the buffer on which the concurrent `readb` operates are distinct. Intuitively, the invariant is maintained by ensuring that there is exactly one owner for each buffer at any time. The main loop transfers ownership of the buffers to the tasks it creates and regains ownership when the tasks terminate.

Our type system explicitly tracks resource ownership and transfer. As in concurrent separation logic, resources describe the *ownership* of heap cells by tasks. The central idea of the type system is that at any point in time, each resource is owned by at most one task. This is implemented by explicit notions of resource ownership and resource transfer.

Ownership and transfer. In the judgment $\Gamma; \eta \vdash e : \varphi$, the task executing e owns the resources in η, meaning that for any resource in η, no other existing task will try to access this resource. When a task p_1 creates a new task p_2, p_1 may relinquish ownership of some of its resources and pass them to p_2; this is known as resource transfer. Conversely, when task p_1 waits for task p_2 to finish, it may also acquire the resources that p_2 holds.

In the double-buffered copying loop example, multiple resource transfers take place. Consider the following slice e_{once} of the code:

```
let task = post (writeb outs buf2) in readb ins buf1; wait task
```

Suppose this code executes in task p_1, and the task created by the **post** statement is p_2. Suppose further that `buf1` has type $\mathsf{ref}_{\mu_1}\,\texttt{buffer}$ and `buf2` has type $\mathsf{ref}_{\mu_2}\,\{\nu : \texttt{buffer} \mid \text{odd}\ \nu\}$. Initially, p_1 has ownership of μ_1 and μ_2. After executing the **post** statement, p_1 passes ownership of μ_2 to p_2 and keeps ownership of μ_1. After executing **wait** `task`, p_1 retains ownership of μ_1, but also regains μ_2 from the now-finished p_2.

Wait permissions. The key idea to ensure that resource transfer is performed correctly is to use *wait permissions*. A wait permission is of the form $\mathsf{Wait}(\pi, \eta)$. It complements a promise by stating which resources (namely the resource set η) may be gained from the terminating task identified by the name π. In contrast to promises, a wait permission may only be used once, to avoid resource duplication. In the following, we use the abbreviations $\mathsf{B} := \texttt{buffer}$ and $\mathsf{B}_{\mathsf{odd}} := \{\nu : \texttt{buffer} \mid \text{odd}\ \nu\}$. Consider again the code slice e_{once} from above. Using ALS types, it types as follows:

$$\Gamma; \mu_r \mapsto \mathsf{B} * \mu_w \mapsto \mathsf{B}_{\mathsf{odd}} \vdash e_{\text{once}} : \varphi \text{ with } \varphi = \mathsf{unit}\langle \mu_r \mapsto \mathsf{B}_{\mathsf{odd}} * \mu_w \mapsto \mathsf{B}_{\mathsf{odd}} \rangle$$

To illustrate the details of resource transfer, consider a slice of e_{once} where the preconditions have been annotated as comments:

ρ	Refinement expressions	$\tau ::= \{\nu : \beta \mid \rho\} \mid x{:}\, \tau\langle\eta\rangle \to \varphi \mid \mathsf{ref}_\mu\, \tau \mid \mathsf{promise}_\pi\, \tau \mid \forall\xi.\tau$
β	Base types	$\eta ::= \mathsf{emp} \mid \mu \mapsto \tau \mid \mathsf{Wait}(\pi, \eta) \mid \eta * \eta$
μ, π, ξ	Resource names	$\varphi ::= И\Xi.\, \tau\langle\eta\rangle$
$\Xi ::= \cdot \mid \Xi, \xi$		$\Gamma ::= \cdot \mid \Gamma, x : \tau \mid \Gamma, \xi$

Figure 3 Syntax of ALS types.

```
(* μ_w ↦ B_odd * μ_r ↦ B*)
let drain_bufw = post (writeb outs bufw) in
(* Wait(π_w, μ_w ↦ B_odd) * μ_r ↦ B*)
let fill_bufr = post (readb ins bufr) in
(* Wait(π_w, μ_w ↦ B_odd) * Wait(π_r, μ_r ↦ B_odd)*)
wait drain_bufw;
(* μ_w ↦ B_odd * Wait(π_r, μ_r ↦ B_odd)*)
wait fill_bufr;
(* μ_w ↦ B_odd * μ_r ↦ B_odd*)
loop bufw bufr
```

Note how the precondition of **wait** `drain_bufw` contains a wait permission $\mathsf{Wait}(\pi_r, \mu_r \mapsto \mathsf{B}_{\mathsf{odd}})$. The resource set $\mu_r \mapsto \mathsf{B}_{\mathsf{odd}}$ describes the postcondition of `readb ins bufr`, and this is the resource set that will be returned by a **wait**.

2.6 Detecting concurrency pitfalls

We now indicate how our type system catches common errors. Consider the following incorrect code that has a race condition:

```
let task = post (writeb outs buf1) in readb ins buf1; wait task
```

Suppose `buf1` types as $\mathsf{ref}_\mu\, \mathsf{B}$. For the code to type check, both p_1 and p_2 would have to own μ. This is, however, not possible by the properties of resource transfer because resources cannot be duplicated. Thus, our type system rejects this incorrect program.

Similarly, suppose the call to the main loop incorrectly passed the same buffer twice: `loop buf1 buf1`. Then, `loop buf1 buf1` would have to be typed with precondition $\mu_1 \mapsto \mathsf{B} * \mu_1 \mapsto \mathsf{B}_{\mathsf{odd}}$. But this resource set is not well-formed, so this code does not type check.

Finally, suppose the order of the buffers was swapped in the initial call to the loop: `loop buf1 buf2`. Typing `loop buf1 buf2` requires a precondition $\mu_1 \mapsto \mathsf{B}_{\mathsf{odd}} * \mu_2 \mapsto \mathsf{B}$. But previous typing steps have established that the precondition will be $\mu_1 \mapsto \mathsf{B} * \mu_2 \mapsto \mathsf{B}_{\mathsf{odd}}$, and even by subtyping, these two resource sets could be made to match only if $\ldots \vdash \mathsf{B} \preceq \mathsf{B}_{\mathsf{odd}}$. But since this is not the case, the buggy program will again not type check.

3 The Type System

We now describe the type system formally. The ALS type system has two notions of types: *value types* and *full types* (see Figure 3). Value types, τ, express the (effect-free) types that values have, whereas full types, φ, are used to type expressions: they describe the type of the computed value and also the heap and task state at the end of the computation.

In order to describe (the local view of) the mutable state of a task, we use *resource sets*, denoted by η, which describe the set of resource names owned by the task. A resource name associates an identifier with physical resources (e.g., heap cells or task ids) that uniquely

identifies it in the context of a typing judgment. In the type system, ξ, μ, and π stand for resource names. We use μ for resource names having to do with heap cells, π for resource names having to do with tasks, and ξ where no distinction is made. Resource names are distinct from "physical names" like pointers to heap cells and task handles. This is needed to support situations in which a name can refer to more than one object, for example, when typing weak references that permit aliasing.

There are five cases for value types τ:

1. Base types $\{\nu : \beta \mid \rho\}$ are type refinements over primitive types β with refinement ρ. Their interpretation is as in liquid types [28].
2. Reference types $\mathsf{ref}_\mu\,\tau$ stand for references to a heap cell that contains a value whose type is a subtype of τ. The type is indexed by a parameter μ, which is a resource name identifying the heap cell.
3. Promise types $\mathsf{promise}_\pi\,\tau$ stand for promises [15] of a value τ. A promise type can be forced, using **wait**, to yield a value of type τ.
4. Arrow types of the form $x{:}\,\tau\langle\eta\rangle \to \varphi$ stand for types of function that may have side effects, and summarize both the interface and the possible side effects of the function. In particular, a function of the above form takes one argument x of (value) type τ. If executed in a global state that matches resource set η, it will, if it terminates, yield a result of full type φ.
5. Resource quantifications of the form $\forall\xi.\tau$ provide polymorphism of names. The type $\forall\xi.\tau$ can be instantiated to any type $\tau[\xi'/\xi]$, as long as this introduces no resource duplications.

Next, consider full types. A full type $\varphi = \text{И}\Xi.\,\tau\langle\eta\rangle$ consists of three parts that describe the result of a computation: a list of resource name bindings Ξ, a value type τ, and a resource set η (introduced below). If an expression e is typed with φ, this means that if it reduces to a value, that value has type τ, and the global state matches η. The list of names Ξ describes names that are allocated during the reduction of e and occur in τ or η. The operator И acts as a binder; each element of Ξ is to be instantiated by a fresh resource name.

Finally, consider resource sets η. Resource sets describe the heap cells and wait permissions owned by a task. They are given in a separation logic notation and consists of a separating conjunction of points-to facts and wait permissions. Points-to facts are written as $\mu \mapsto \tau$ and mean that for the memory location(s) associated with the resource name μ, the values residing in those memory locations can be typed with value type τ, similar to Alias Types [32]. The resource emp describes that no heap cells or wait permissions are owned. Conjunction $\eta_1 * \eta_2$ means that the resources owned by a task can be split into two disjoint parts, one described by η_1 and the other by η_2. The notion of disjointness is given in terms of the *name sets* of η: The name set of η is defined as $\mathsf{Names}(\mathsf{emp}) = \varnothing$, $\mathsf{Names}(\mu \mapsto \tau) = \{\mu\}$ and $\mathsf{Names}(\eta_1 * \eta_2) = \mathsf{Names}(\eta_1) \cup \mathsf{Names}(\eta_2)$. The resources owned by η are then given by $\mathsf{Names}(\eta)$, and the resources of η_1 and η_2 are disjoint iff $\mathsf{Names}(\eta_1) \cap \mathsf{Names}(\eta_2) = \varnothing$.

A resource of the form $\mathsf{Wait}(\pi, \eta)$ is called a *wait permission.* A wait permission describes the fact that the process indicated by π will hold the resources described by η upon termination, and the owner of the wait permissions may acquire theses resources by waiting for the task. Wait permissions are used to ensure that no resource is lost or duplicated in creating and waiting for a task, and to carry out resource transfers. For wellformedness, we demand that $\pi \notin \mathsf{Names}(\eta)$, and define $\mathsf{Names}(\mathsf{Wait}(\pi, \eta)) = \mathsf{Names}(\eta) \cup \{\pi\}$.

Lastly, $*$ is treated as an associative and commutative operator with unit emp, and resources that are the same up to associativity and commutativity are identified. For example, $\mu_1 \mapsto \tau_1 * \mathsf{Wait}(\pi, \mu_2 \mapsto \tau_2 * \mu_3 \mapsto \tau_3)$ and $\mu_1 \mapsto \tau_1 * \mathsf{Wait}(\pi, \mu_3 \mapsto \tau_3 * (\mathsf{emp} * \mu_2 \mapsto \tau_2))$ are considered the same resource.

$$\frac{[\![\Gamma]\!] \models \forall \nu.[\![\rho_1]\!] \implies [\![\rho_2]\!] \quad \Gamma \vdash \{\nu : \beta \mid \rho_1\} \text{ wf} \quad \Gamma \vdash \{\nu : \beta \mid \rho_2\} \text{ wf}}{\Gamma \vdash \{\nu : \beta \mid \rho_1\} \preceq \{\nu : \beta \mid \rho_2\}} \qquad \frac{\Gamma \vdash \tau_2 \preceq \tau_1 \quad \Gamma, x : \tau_2 \vdash \eta_2 \preceq \eta_1 \quad \Gamma, x : \tau_2 \vdash \varphi_1 \preceq \varphi_2}{\Gamma \vdash x{:}\,\tau_1\langle\eta_1\rangle \to \varphi_1 \preceq x{:}\,\tau_2\langle\eta_2\rangle \to \varphi_2}$$

$$\frac{\Gamma \vdash \tau \text{ wf}}{\Gamma \vdash \tau \preceq \tau} \qquad \frac{\Gamma, \xi \vdash \tau_1 \preceq \tau_2}{\Gamma \vdash \forall \xi.\tau_1 \preceq \forall \xi.\tau_2} \qquad \frac{\Gamma \vdash \tau_1 \preceq \tau_2 \quad \Gamma \vdash \mu}{\Gamma \vdash \mu \mapsto \tau_1 \preceq \mu \mapsto \tau_2} \qquad \frac{\vdash \Gamma \text{ wf}}{\Gamma \vdash \eta \preceq \eta}$$

$$\frac{\Gamma \vdash \eta_1 \preceq \eta_1' \quad \Gamma \vdash \eta_2 \preceq \eta_2' \quad \Gamma \vdash \eta_1 * \eta_2 \text{ wf} \quad \Gamma \vdash \eta_1' * \eta_2' \text{ wf}}{\Gamma \vdash \eta_1 * \eta_2 \preceq \eta_1' * \eta_2'} \qquad \frac{\Xi \subseteq \Xi' \quad \Gamma, \Xi \vdash \tau_1 \preceq \tau_2 \quad \Gamma, \Xi \vdash \eta_1 \preceq \eta_2}{\Gamma \vdash \text{И}\Xi.\,\tau_1\langle\eta_1\rangle \preceq \text{И}\Xi'.\,\tau_2\langle\eta_2\rangle}$$

Figure 4 Subtyping rules. The notations $[\![\cdot]\!]$ and $\models$ are defined in [28].

3.1 Typing rules

The connection between expressions and types is made using the typing rules of the core calculus. The typing rules use auxiliary judgments to describe wellformedness and subtyping. There are four types of judgments used in the type system: wellformedness, subtyping, value typing and expression typing. Wellformedness provides three judgments, one for each kind of type: wellformedness of value types $\Gamma \vdash \tau$ wf, of resources $\Gamma \vdash \eta$ wf and of full types $\Gamma \vdash \varphi$ wf. Subtyping judgments are of the form $\Gamma \vdash \tau_1 \preceq \tau_2$, $\Gamma \vdash \eta_1 \preceq \eta_2$ and $\Gamma \vdash \varphi_1 \preceq \varphi_2$. Finally, value typing statements are of the form $\Gamma \vdash v : \tau$, while expression typing statements are of the form $\Gamma; \eta \vdash e : \varphi$.

The typing environment Γ is a list of variable bindings of the form $x : \tau$ and resource name bindings ξ. We assume that all environments are wellformed, i.e., no name or variable is bound twice and in all bindings of the form $x : \tau$, the type τ is wellformed.

The wellformedness rules are straightforward; details can be found in the full version [14]. They state that all free variables in a value type, resource or full type are bound in the environment, and that no name occurs twice in any resource, i.e., for each subexpression $\eta_1 * \eta_2$, the names in η_1 and η_2 are disjoint, and for each subexpression $\mathsf{Wait}(\pi, \eta)$, we have $\pi \notin \mathsf{Names}(\eta)$.

The subtyping judgments are defined in Figure 4. Subtyping judgments describe that a value, resource, or full type is a subtype of another object of the same kind. Subtyping of base types is performed by semantic subtyping of refinements (i.e., by logical implication), as in liquid types [28]. References are invariant under subtyping to ensure type safety.

Arrow type subtyping follows the basic pattern of function type subtyping: arguments – including the resources – are subtyped contravariantly, while results are subtyped covariantly.

Resource subtyping is performed pointwise: $\Gamma \vdash \eta_1 \preceq \eta_2$ holds if the wait permissions in η_1 are the same as in η_2, if μ points to τ_1 in η_1, then it points to τ_2 in η_2 where $\Gamma \vdash \tau_1 \preceq \tau_2$, and if μ points to τ_2 in η_2, it points to some τ_1 in η_1 with $\Gamma \vdash \tau_1 \preceq \tau_2$.

3.2 Value and expression typing

Figure 5 shows some of the value and expression typing rules. Value typing, $\Gamma \vdash v : \tau$, assigns a value type τ to a value v in the environment Γ, whereas expression typing, $\Gamma; \eta \vdash e : \varphi$ assigns, given an initial resource η, called the *precondition*, and an environment Γ, a full type φ to an expression e. The value typing rules and the subtyping rules are standard, and typing a value as an expression gives them types as an effect-free expression: From an

$$\textsc{TV-Const}\ \dfrac{\vdash \Gamma \text{ wf}}{\Gamma \vdash c : \text{typeof}(c)} \qquad \textsc{TV-Var}\ \dfrac{\vdash \Gamma \text{ wf} \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{TV-Lambda}\ \dfrac{\Gamma, x : \tau; \eta \vdash e : \varphi}{\Gamma \vdash \lambda x.e : x{:}\,\tau\langle\eta\rangle \to \varphi}$$

$$\textsc{T-Value}\ \dfrac{\Gamma \vdash v : \tau}{\Gamma; \mathsf{emp} \vdash v : \tau\langle\mathsf{emp}\rangle} \qquad \textsc{TV-Subtype}\ \dfrac{\Gamma \vdash \tau \preceq \tau' \qquad \Gamma \vdash v : \tau}{\Gamma \vdash v : \tau'} \qquad \textsc{T-Subtype}\ \dfrac{\Gamma \vdash \varphi \preceq \varphi' \qquad \Gamma; \eta \vdash e : \varphi}{\Gamma; \eta \vdash e : \varphi'}$$

$$\textsc{TV-ForallIntro}\ \dfrac{\Gamma, \xi \vdash v : \tau}{\Gamma \vdash v : \forall \xi.\tau} \qquad \textsc{T-ForallElim}\ \dfrac{\Gamma; \eta \vdash e : \text{И}\Xi.\, \forall\xi.\tau\langle\eta'\rangle \qquad \Gamma \vdash \text{И}\Xi.\, \tau[\xi'/\xi]\langle\eta'\rangle \text{ wf}}{\Gamma; \eta \vdash e : \text{И}\Xi.\, \tau[\xi'/\xi]\langle\eta'\rangle}$$

$$\textsc{T-Frame}\ \dfrac{\Gamma; \eta_1 \vdash e : \text{И}\Xi.\, \tau\langle\eta_2\rangle \qquad \Gamma \vdash \eta_1 * \eta \text{ wf} \qquad \Gamma, \Xi \vdash \eta_2 * \eta \text{ wf}}{\Gamma; \eta_1 * \eta \vdash e : \text{И}\Xi.\, \tau\langle\eta_2 * \eta\rangle}$$

$$\textsc{T-Ref}\ \dfrac{\Gamma; \eta_1 \vdash e : \text{И}\Xi.\, \tau\langle\eta_2\rangle \qquad \Gamma, \Xi \vdash \tau' \preceq \tau \qquad \mu \text{ fresh resource name variable}}{\Gamma; \eta_1 \vdash \mathbf{ref}\, e : \text{И}\Xi, \mu.\, \mathsf{ref}_\mu\, \tau'\langle\eta_2 * \mu \mapsto \tau\rangle} \qquad \textsc{T-WRef}\ \dfrac{\Gamma; \eta_1 \vdash e : \text{И}\Xi.\, \tau\langle\eta_2\rangle \qquad \mu \text{ weak}}{\Gamma; \eta_1 \vdash \mathbf{ref}\, e : \text{И}\Xi, \mu.\, \mathsf{ref}_\mu\, \tau'\langle\eta_2\rangle}$$

$$\textsc{T-Read}\ \dfrac{\Gamma; \eta_1 \vdash e : \text{И}\Xi.\, \mathsf{ref}_\mu\, \tau'\langle\eta_2 * \mu \mapsto \tau\rangle}{\Gamma; \eta_1 \vdash\, !e : \text{И}\Xi.\, \tau\langle\eta_2 * \mu \mapsto \tau\rangle} \qquad \textsc{T-WRead}\ \dfrac{\Gamma; \eta_1 \vdash e : \text{И}\Xi.\, \mathsf{ref}_\mu\, \tau\langle\eta_2\rangle \qquad \mu \text{ weak}}{\Gamma; \eta_1 \vdash\, !e : \text{И}\Xi.\, \tau\langle\eta_2\rangle}$$

$$\textsc{T-Write}\ \dfrac{\Gamma; \eta_1 \vdash e_2 : \text{И}\Xi_1.\, \tau_2\langle\eta_2\rangle \qquad \Gamma, \Xi_1, \Xi_2 \vdash \tau_2 \preceq \tau \qquad \Gamma, \Xi_1; \eta_2 \vdash e_1 : \text{И}\Xi_2.\, \mathsf{ref}_\mu\, \tau\langle\eta_3 * \mu \mapsto \tau_1\rangle}{\Gamma; \eta_1 \vdash e_1 := e_2 : \text{И}\Xi_1, \Xi_2.\, \mathsf{unit}\langle\eta_3 * \mu \mapsto \tau_2\rangle}$$

$$\textsc{T-WWrite}\ \dfrac{\Gamma; \eta_1 \vdash e_2 : \text{И}\Xi_1.\, \tau\langle\eta_2\rangle \qquad \Gamma, \Xi_1; \eta_2 \vdash e_1 : \text{И}\Xi.\, \mathsf{ref}_\mu\, \tau\langle\eta_3\rangle \qquad \mu \text{ weak}}{\Gamma; \eta_1 \vdash e_1 := e_2 : \text{И}\Xi.\, \mathsf{unit}\langle\eta_3\rangle}$$

$$\textsc{T-Post}\ \dfrac{\Gamma; \eta \vdash e : \text{И}\Xi.\, \tau\langle\eta'\rangle \qquad \pi \text{ fresh resource name variable}}{\Gamma; \eta \vdash \mathbf{post}\, e : \text{И}\Xi, \pi.\, \mathsf{promise}_\pi\, \tau\langle\mathsf{Wait}(\pi, \eta')\rangle} \qquad \textsc{T-WaitTransfer}\ \dfrac{\Gamma; \eta \vdash e : \text{И}\Xi.\, \mathsf{promise}_\pi\, \tau\langle\eta_1 * \mathsf{Wait}(\pi, \eta_2)\rangle}{\Gamma; \eta \vdash \mathbf{wait}\, e : \text{И}\Xi.\, \tau\langle\eta_1 * \eta_2\rangle}$$

$$\textsc{T-App}\ \dfrac{\Gamma; \eta_1 \vdash e_2 : \text{И}\Xi_1.\, \tau_x\langle\eta_2\rangle \qquad \Gamma, \Xi_1; \eta_2 \vdash e_1 : \text{И}\Xi_2.\, (x{:}\, \tau_x\langle\eta_4\rangle \to \text{И}\Xi_3.\, \tau\langle\eta_5\rangle)\langle\eta_3\rangle \qquad \Gamma, \Xi_1 \vdash \eta_3 \preceq \eta_4 * \eta_i \qquad \Gamma \vdash \text{И}\Xi_1, \Xi_2, \Xi_3.\, \tau\langle\eta_5 * \eta_i\rangle \text{ wf}}{\Gamma; \eta_1 \vdash e_1\, e_2 : \text{И}\Xi_1, \Xi_2, \Xi_3.\, \tau\langle\eta_5 * \eta_i\rangle}$$

Figure 5 Value and expression typing.

empty precondition η, they yield a result of type τ with empty postcondition and no name allocation.

The rules TV-ForallIntro and T-ForallElim allow for the quantification of resource names for function calls. This is used to permit function signatures that are parametric in the resource names, and can therefore be used with arbitrary heap and task handles as arguments. The typing rules are based on the universal quantification rules for Indexed Types [37], and are similar to the quantification employed in alias types.

The rule T-Frame implements the frame rule from separation logic [27] in the context of ALS types. It allows adjoining a resource that is left invariant by the execution of an expression e to the pre-condition and the post-condition.

$$e_1\, e_2 \hookrightarrow e_1\, v \hookrightarrow (\lambda x.e)\, v \hookrightarrow e[v/x] = e[v/x] \hookrightarrow v'$$

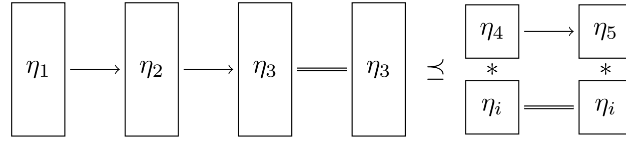


Figure 6 Transformation of the global state as modeled by the T-App rule. Upper row: expression reduction steps, lower row: The corresponding resources of the global state.

The typing rules T-Ref, T-Read and T-Write type the memory access operations. The typing rules implement strong heap updates using the pre- and post-conditions. This is possible because separate resources for pre- and post-conditions that are tied to specific global states are used, whereas the type of the reference only describes an upper bound for the type of the actual cell contents. A similar approach is used in low-level liquid types [29]. Additionally, the rules T-WRef, T-WRead and T-WWrite allow for weak heap updates, using a subset of locations that is marked as weak and never occur in points-to facts.

It is important to note how the evaluation order affects these typing rules. For example, when evaluating e_1 `:=` e_2, we first reduce to e_1 `:=` v, and then to ℓ `:=` v. Therefore T-Write types e_2 with the initial η_1 precondition and uses the derived postcondition, η_2, as a precondition for typing e_1.

The typing rules T-Post and T-WaitTransfer serve the dual purpose of providing the proper return type to the concurrency primitives (`post` and `wait`) and to control the transfer of resource ownership between tasks.

T-Post types task creation using an expression of the form **post** e. For an expression e that yields a value of type τ and a resource η, it gives a promise that if evaluating the expression e terminates, waiting for the task will yield a value of type τ, and additionally, if some task acquires the resources of the task executing e, it will receive exactly the resources described by η.

T-WaitTransfer types expressions that wait for the termination of a task with resource transfer. It states that if e returns a promise for a value τ and a corresponding wait permission $\mathsf{Wait}(\pi, \eta_2)$ yielding a resource η_2, as well as some additional resource η_1, then **wait** e yields a value of type τ, and the resulting global state has a resource $\eta_1 * \eta_2$. In particular, the postcondition describes the union of the postcondition of e, without the wait permission $\mathsf{Wait}(\pi, \eta_2)$, and the postcondition of the task that e refers to, as given by the wait permission.

Finally, T-App types function applications under the assumption that the expression is evaluated from right to left, as in OCaml. The first two preconditions on the typing of e_1 and e_2 are standard up to the handling of resources, while the wellformedness condition ensures that the variable x does not escape its scope. The resource manipulation of T-App is illustrated in Fig. 6. Resources are chosen in such a way that they describe the state transformation of first reducing e_2 to a value, then e_1 and finally the β-redex of $e_1 e_2$.

The type system contains several additional rules for handling if–then–else expressions and for dealing with weak references. These rules are completely standard and can be found in the full version.

3.3 Type safety

The type system presented above enjoys type safety in terms of a global typing relation. The details can be found in the full version; here, only the notion of global typing and the type safety statement are sketched.

We need the following three functions. The *global type* γ is a function that maps heap locations to value types and task identifiers to full types. For heap cells, it describes the type of the reference to that heap cell, and for a task, the postcondition type of the task. The *global environment* ψ is a function that maps heap locations to value types and task identifiers to resources. For heap cells, it describes the precise type of the cell content, and for a task, the precondition of the task. The *name mapping* χ is a function that maps heap locations and task identifiers to names. It is used to connect the heap cells and tasks to their names used in the type system.

For the statement of type safety, we need three definitions:

1. Given γ, ψ and χ, we say that γ, ψ and χ type a global configuration, written $\psi, \chi \vdash (H, P, p) : \gamma$, when:
   - For all $\ell \in \operatorname{dom} H$, $\Gamma_\ell \vdash H(\ell) : \psi(\ell)$,
   - For all $p \in \operatorname{dom} P$, $\Gamma_p; \psi(p) \vdash P(p) : \gamma(p)$

   where the Γ_ℓ and Γ_p environments are defined in the full version. In other words, the heap cells can be typed with their current, precise type, as described by ψ, while the tasks can be typed with the type give by γ, using the precondition from ψ.
2. γ, ψ and χ are *well-formed*, written (γ, ψ, χ) wf, if a number of conditions are fulfilled. The intuition is that on one hand, a unique view of resource ownership can be constructed from the three functions, and on the other hand, different views of resources (e.g., the type of a heap cell as given by a precondition compared with the actual type of the heap cell) are compatible.
3. For two partial functions f and g, f extends g, written $g \sqsubseteq f$, if $\operatorname{dom} g \subseteq \operatorname{dom} f$ and $f(x) = g(f)$ for all $x \in \operatorname{dom} g$.
   Given two global type γ and γ', and two name maps χ and χ', we say that (γ, χ) *specializes to* (γ', χ'), written $(\gamma, \chi) \rhd (\gamma', \chi')$, when the following holds: $\chi \sqsubseteq \chi'$, $\gamma\ |_{\mathsf{Locs}} \sqsubseteq \gamma'\ |_{\mathsf{Locs}}$, $\operatorname{dom} \gamma \subseteq \operatorname{dom} \gamma'$ and for all task identifiers $p \in \operatorname{dom} \gamma$, $\gamma'(p)$ specializes γ in the following sense: Let $\varphi = \text{И}\Xi.\, \tau\langle\eta\rangle$ and $\varphi' = \text{И}\Xi'.\, \tau'\langle\eta'\rangle$ be two full types. Then φ' specializes φ if there is a substitution σ such that $\text{И}\Xi'.\, \tau\sigma\langle\eta\sigma\rangle = \varphi'$, i.e., φ' can be gotten from φ by instantiating some names.

The following theorem follows using a standard preservation/progress argument.

▶ **Theorem 1** (Type safety). *Consider a global configuration* (H, P, p) *that is typed as* $\psi, \chi \vdash (H, P, p) : \gamma$*. Suppose that* (γ, ψ, χ) *wf.*

Then for all (H', P', p') *such that* $(H, P, p) \hookrightarrow^* (H', P', p)$*, there are* γ', ψ', χ' *such that* $\psi', \chi' \vdash (H', P', p') : \gamma'$*,* (γ', ψ', χ') *wf and* $(\gamma, \psi) \rhd (\gamma', \psi')$*.*

Furthermore, if (H', P', p') *cannot take a step, then all processes in* P' *have terminated, in the sense that the expressions of all tasks have reduced to values.*

4 Type Inference

To infer ALS types, we extend the liquid type inference algorithm. The intention was to stay as close to the original algorithm as possible. The liquid type inference consists of the four steps depicted on the left of Figure 7:

1. Basic typing assigns plain OCaml types to the expression that is being typed using the Hindley-Milner algorithm.
2. The typing derivation is processed to add refinements to the types. In those cases where a clear refinement is known (e.g., for constants), that refinement is added to the type. In all other cases, a refinement variable is added to the type. In the latter case, additional constraints are derived that limit the possible instantiations of the refinement variables.

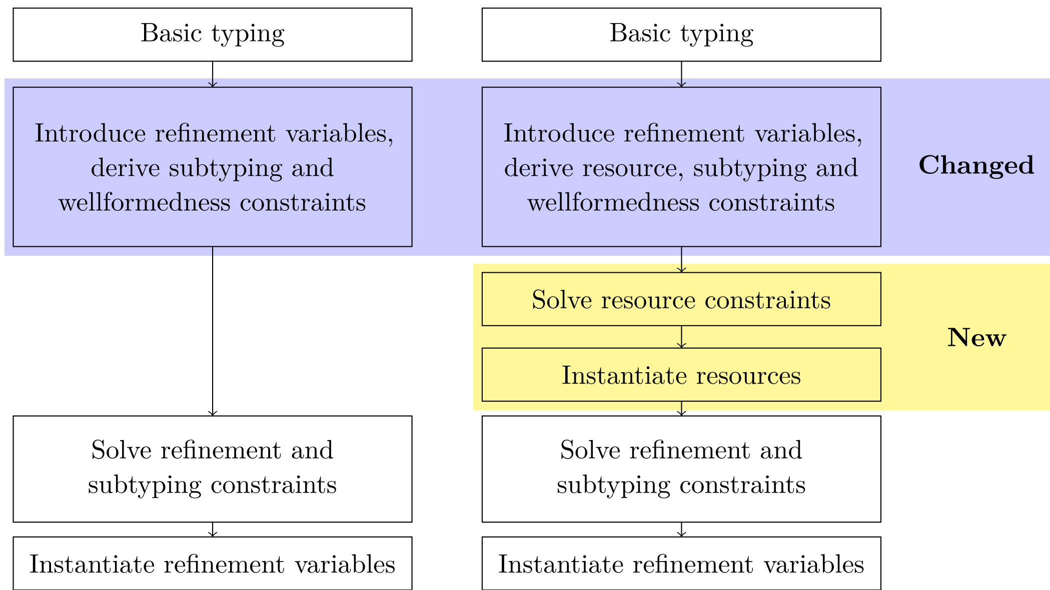


Figure 7 High-level overview of the liquid type inference procedure (left), and the modified procedure presented in this section (right). The changes are highlighted.

For example, consider the typing of an application $e_1 e_2$. Suppose e_1 has the refined type $x : \{\nu : \texttt{int} \mid \nu \geq 0\} \to \{\nu : \texttt{int} \mid \nu = x + 1\}$, and e_2 has refined type $\{\nu : \texttt{int} \mid \nu \geq 5\}$. From step 1, $e_1 e_2$ has type `int`. In this step, this type is augmented to $\{\nu : \texttt{int} \mid \rho\}$, where ρ is a refinement variable, and two constraints are produced:

- $\vdash x : \{\nu : \texttt{int} \mid \nu \geq 0\} \to \{\nu : \texttt{int} \mid \nu = x + 1\} \preceq x : \{\nu : \texttt{int} \mid \nu \geq 5\} \to \{\nu : \texttt{int} \mid \rho\}$, describing that the function type should be specialized taking the more precise type of the argument into account,
- $\vdash \{\nu : \texttt{int} \mid \rho\}$ wf, describing that $\{\nu : \texttt{int} \mid \rho\}$ should be well-formed. In particular, the instantiation of ρ may not mention the variable x.

3. The constraints from the second step are solved relative to a set of user-provided predicates. In the example, one possible solution for ρ would be $\nu \geq 6$.
4. The solutions from the third step are substituted for the refinement variables. In the example, $e_1 e_2$ would therefore get the type $\{\nu : \texttt{int} \mid \nu \geq 6\}$.

The details of this procedure are described in [28]. For ALS types, the procedure is extended to additionally derive the resources that give preconditions and postconditions for the expressions. This involves a new type of variables, *resource variables*, which are placeholders for pre- and post-conditions. This is depicted on the right-hand side of Figure 7.

Several steps are identical to the algorithm above; the constraint derivation step has been modified, whereas the steps dealing with resource variables are new. We sketch the working of the algorithm by way of a small example. Consider the following expression:

```
let x = post (ref 1) in !(wait x)
```

After applying basic typing, the expression and its sub-expressions can be typed as follows:

```
let x = post ( ref ( 1   ) ) in  !( wait( x                ) )
                     int                   promise (ref int)
               ref int               ref int
        promise (ref int)         int
int
```

The second step then derives the following ALS typing derivation. In this step, each precondition and postcondition gets a new resource variable:

$$\begin{array}{l}
\textbf{let}\ \mathtt{x} = \textbf{post}\ (\ \textbf{ref}\ (\ \mathtt{1}\ )\ )\ \textbf{in}\ \ !(\ \textbf{wait}(\ \mathtt{x}\ )\ ) \\
\quad \eta_2 \Rightarrow \mathtt{int}_{=1}\langle \eta_2 \rangle \qquad\qquad \eta_4 \Rightarrow \tau_x\langle \eta_4 \rangle \\
\quad \eta_2 \Rightarrow \text{И}\xi_1.\,\mathsf{ref}_{\xi_1}\ \mathtt{int}_{\rho_1}\langle \eta_3 \rangle \qquad \eta_4 \Rightarrow \mathsf{ref}_{\xi_1}\ \mathtt{int}_{\rho_1}\langle \eta_5 \rangle \\
\quad \eta_1 \Rightarrow \text{И}\xi_2.\,\underbrace{\mathsf{promise}_{\xi_2}\ (\mathsf{ref}_{\xi_1}\ \mathtt{int}_{\rho_1})}_{\tau_x}\langle \eta_4 \rangle \qquad \eta_4 \Rightarrow \mathtt{int}_{\rho_2}\langle \eta_6 \rangle \\
\eta_1 \Rightarrow \mathtt{int}_{\rho_2}\langle \eta_6 \rangle
\end{array}$$

Here, an expression e types as $\eta \Rightarrow \text{И}\Xi.\,\tau\langle\eta'\rangle$ iff, for some environment Γ and some Ξ', $\Gamma; \eta \vdash e : \text{И}\Xi, \Xi'.\,\tau\langle\eta'\rangle$. The η_i occurring in the derivation are all variables.

Three types of constraints are derived: subtyping and wellformedness constraints (for the refinement variables), and resource constraints (for the resource variables). For the first two types of constraints, the following constraints are derived:

- $\vdash \mathtt{int}_{=1} \preceq \mathtt{int}_{\rho_1}$, derived from the typing of **ref** `1`: The reference type $\mathtt{int}_{\rho_1}$ must allow a cell content of type $\mathtt{int}_{=1}$.
- $x : \tau_x \vdash \mathtt{int}_{\rho_2} \preceq \mathtt{int}_{\rho_1}$, derived from the typing of `(`**wait** `x)`: The cell content type of the cell ξ_1 must be a subtype of the type of the reference.
- $\vdash \mathtt{int}_{\rho_2}$ wf, which derives from the **let** expression: The type $\mathtt{int}_{\rho_2}$ must be well-formed outside the **let** expression, and therefore, must not contain the variable x.

The refinement constraints can be represented by a constraint graph representing heap accesses, task creation and finalization, and function calls. For the example, we get the following constraint graph:

$$\begin{array}{ccccccc}
\eta_2 & \xrightarrow{\text{Alloc}(\xi_1,\ \mathtt{int}_{\rho_1})} & \eta_3 & & & & \\
\uparrow & & \downarrow & & & & \\
\eta_1 & \xrightarrow{\text{Post}(\xi_2)} & \eta_4 & \xrightarrow{\text{Wait}(\xi_2)} & \eta_5 & \xrightarrow{\text{Read}(\xi_1,\ \mathtt{int}_{\rho_2})} & \eta_6
\end{array}$$

Here, $\text{Alloc}(\xi_1,\ \mathtt{int}_{\rho_1})$ stands for "Allocate a cell with name ξ_1 containing data of type $\mathtt{int}_{\rho_1}$" and so on.

To derive the correct resources for the resource variables, we make use of the following observation. Given an η, say, $\eta = \mathbf{wait}(\pi_1, \mathbf{wait}(\pi_2, \mu \mapsto \tau))$, each name occurring in this resource has a unique sequence of task names π associated with it that describe in which way it is enclosed by wait permissions. This sequence is called its *wait prefix*. In the example, μ is enclosed in a wait permissions for π_2, which is in turn enclosed by one for π_1, so the wait prefix for μ is $\pi_1\pi_2$. For π_2, it is π_1, while for π_1, it is the empty sequence ϵ.

It is easy to show that a resource η can be uniquely reconstructed from the wait prefixes for all the names occurring in η, and the types of the cells occurring in η. In the inference algorithm, the wait prefixes and the cell types for each resource variable are derived independently.

First, the algorithm derives wait prefixes for each refinement variable by applying abstract interpretation to the constraint graph. For this, the wait prefixes are embedded in a lattice $\text{Names} \to \{U, W\} \cup \{p \mid p \text{ prefix}\}$, where $U \sqsubset p \sqsubset W$ for all prefixes p. Here, U describes that a name is unallocated, whereas W describes that a name belongs to a weak reference.

In the example, the following mapping is calculated:

$$\begin{array}{ccccccc}
\eta_2 : \bot, \bot & \xrightarrow{\text{Alloc}(\xi_1,\ \mathtt{int}_{\rho_1})} & \eta_3 : \epsilon, \bot & & & & \\
\uparrow & & \downarrow & & & & \\
\eta_1 : \bot, \bot & \xrightarrow{\text{Post}(\xi_2)} & \eta_4 : \xi_2, \epsilon & \xrightarrow{\text{Wait}(\xi_2)} & \eta_5 : \epsilon, \bot & \xrightarrow{\text{Read}(\xi_1,\ \mathtt{int}_{\rho_2})} & \eta_6 : \epsilon, \bot
\end{array}$$

The mapping is read as follows: If $\eta : w_1, w_2$, then ξ_1 has wait prefix w_1 and ξ_2 has wait prefix w_2.

In this step, several resource usage problems can be detected:

- A name corresponding to a wait permission is not allowed to be weak, because that would mean that there are two tasks sharing a name, which would break the resource transfer semantics.
- When waiting for a task with name π, π must have prefix ϵ: The waiting task must possess the wait permission.
- When reading or writing a heap cell with name μ, μ must have prefix ϵ or be weak, by a similar argument.

Second, the algorithm derives cell types for each refinement variable. This is done by propagating cell types along the constraint graph; if a cell can be seen to have multiple refinements $\{\nu : \tau \mid \rho_1\}, \ldots, \{\nu : \tau \mid \rho_n\}$, a new refinement variable ρ is generated and subtyping constraints $\Gamma \vdash \{\nu : \tau \mid \rho_1\} \preceq \{\nu : \tau \mid \rho\}, \ldots, \Gamma \vdash \{\nu : \tau \mid \rho_n\} \preceq \{\nu : \tau \mid \rho\}$ are added to the constraint set. In the example, the following mapping is calculated for cell ξ_1 (where $\bot$ stands for "cell does not exist"):

$$\begin{array}{ccccccc}
\eta_2 : \bot & \xrightarrow{\text{Alloc}(\xi_1,\, \mathtt{int}_{\rho_1})} & \eta_3 : \mathtt{int}_{=1} & & & & \\
\uparrow & & \downarrow & & & & \\
\eta_1 : \bot & \xrightarrow{\text{Post}(\xi_2)} & \eta_4 : \mathtt{int}_{=1} & \xrightarrow{\text{Wait}(\xi_2)} & \eta_5 : \mathtt{int}_{=1} & \xrightarrow{\text{Read}(\xi_1,\, \mathtt{int}_{\rho_2})} & \eta_6 : \mathtt{int}_{=1}
\end{array}$$

Additionally, a new subtyping constraint is derived: $x : \tau_x \vdash \mathtt{int}_{=1} \preceq \mathtt{int}_{\rho_2}$. Using this information, instantiations for the resource variables can be computed:

$$\eta_1, \eta_2 : \mathsf{emp} \qquad \eta_3, \eta_5, \eta_6 : \xi_1 \mapsto \mathtt{int}_{=1} \qquad \eta_4 : \mathsf{Wait}(\xi_2, \xi_1 \mapsto \mathtt{int}_{=1})$$

These instantiations are then substituted wherever resource variables occur, both in constraints and in the type of the expression. We get the following type for the expression (using $\eta_f = \xi_1 \mapsto \mathtt{int}_{=1}$, $\eta_w := \mathsf{Wait}(\xi_2, \xi_1 \mapsto \mathtt{int}_{=1})$ and τ_x from above):

let x = **post** (**ref** (1)) **in** !(**wait**(x))

- `1` : $\mathsf{emp} \Rightarrow \mathtt{int}_{=1}\langle \mathsf{emp} \rangle$
- **ref** (1) : $\mathsf{emp} \Rightarrow ИH\xi_1.\, \mathsf{ref}_{\xi_1}\ \mathtt{int}_{\rho_1}\langle \eta_f \rangle$
- **post** (**ref** (1)) : $\mathsf{emp} \Rightarrow И\xi_2.\, \tau_x\langle \eta_w \rangle$
- `x` : $\eta_w \Rightarrow \tau_x\langle \eta_w \rangle$
- **wait**(x) : $\eta_w \Rightarrow \mathsf{ref}_{\xi_1}\ \mathtt{int}_{\rho_1}\langle \eta_f \rangle$
- !(**wait**(x)) : $\eta_w \Rightarrow \mathtt{int}_{\rho_2}\langle \eta_f \rangle$
- whole expression: $\mathsf{emp} \Rightarrow \mathtt{int}_{\rho_2}\langle \eta_f \rangle$

Additionally, some further subtyping and wellformedness constraints are introduced to reflect the relationship between cell types, and to give lower bounds on the types of reads. In the example, one new subtyping constraint is introduced: $x : \tau_x \vdash \mathtt{int}_{=1} \preceq \mathtt{int}_{\rho_2}$, stemming from the read operation $\text{Read}(\xi_1, \mathtt{int}_{\rho_2})$ that was introduced for the reference access $!(\textbf{wait}\ x)$. It indicates that the result of the read has a typing $\mathtt{int}_{\rho_2}$ that subsumes that cell content type, $\mathtt{int}_{=1}$.

At this point, it turns out that, when using this instantiation of resource variables, the resource constraints are fulfilled as soon as the subtyping and wellformedness constraints are fulfilled. The constraints handed to the liquid type constraint solver are:

$$\vdash \mathtt{int}_{=1} \preceq \mathtt{int}_{\rho_1} \qquad x : \tau_x \vdash \mathtt{int}_{\rho_2} \preceq \mathtt{int}_{\rho_1} \qquad \vdash \mathtt{int}_{\rho_2}\ \text{wf} \qquad x : \tau_x \vdash \mathtt{int}_{=1} \preceq \mathtt{int}_{\rho_2}$$

This leads to the instantiation of ρ_1 and ρ_2 with the predicate $\nu = 1$.

5 Case Studies

We have extended the existing liquid type inference tool, `dsolve`, to handle ALS types. Below, we describe our experiences on several examples taken from the literature and real-world code.

In general, the examples make heavy use of external functions. For this reason, some annotation work will always be required. In many cases, it turns out that only few functions will have to be explicitly annotated with ALS types. In the examples, we state how many annotations were used in each case.

Our implementation only supports liquid type annotations on external functions but not ALS types. We work around this by giving specifications of abstract purely functional versions of functions, and providing an explicit wrapper implementation that implement the correct interface. For example, suppose we want to provide the following external function:

$$\texttt{write}\colon \texttt{stream} \to \mathsf{ref}_\xi\, \texttt{buffer}\langle \xi \mapsto \{\nu : \texttt{buffer} \mid \nu \text{ odd}\}\rangle \to (\mathsf{unit}\langle \xi \mapsto \texttt{buffer}\rangle)$$

We implement this by providing an external function

$$\texttt{write_sync}\colon \texttt{stream} \to \{\nu : \texttt{buffer} \mid \nu \text{ odd}\} \to \texttt{buffer}$$

and a wrapper implementation

```
let write s b = b := write_sync s (!b)
```

The wrapper code is counted separately from annotation code.

5.1 The double-buffering example, revisited

Our first example is the double-buffering copy loop from Section 2. We consider three versions of the code:

1. The copying loop, exactly as given.
2. A version of the copying loop in which an error has been introduced. Instead of creating a task that writes a full buffer to the disk, i.e., **post** `(Writer.write outs buffer_full)`, we post a task that tries to write the read buffer: **post**`(Writer.write outs buffer_empty)`.
3. Another version of the copying loop. This time, the initial call to the main loop is incorrect: the buffers are switched, so that the loop would try to write the empty buffer while reading into the full buffer.

We expect the type check to accept the first version of the example and to detect the problems in the other two versions.

We use the following ALS type annotations:

$$\begin{aligned}
\texttt{write} : s : \texttt{stream} &\to \forall \mu.\mathsf{ref}_\mu\, \texttt{buffer}\langle \mu \mapsto \{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}\rangle \to \\
&\quad \mathsf{unit}\langle \mu \mapsto \{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}\rangle \\
\texttt{read} : s : \texttt{stream} &\to \forall \mu.\mathsf{ref}_\mu\, \texttt{buffer}\langle \mu \mapsto \texttt{buffer}\rangle \to \\
&\quad \mathsf{unit}\langle \mu \mapsto \{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}\rangle \\
\texttt{make_buffer} : \mathsf{unit} &\to \text{И}\mu.\, \mathsf{ref}_\mu\, \texttt{buffer}\langle \mu \mapsto \{\nu : \texttt{buffer} \mid \neg\mathrm{odd}(\nu)\}\rangle
\end{aligned}$$

The main use of annotations is to introduce the notion of a buffer with odd parity. Using a predicate `odd`, we can annotate the contents of a buffer cell to state whether it has odd

parity or not. For example, the function `read` has type[3].

$$s : \texttt{stream} \to b : \mathsf{ref}_\xi\,\texttt{buffer}\langle \xi \mapsto \texttt{buffer}\rangle \to \mathsf{unit}\langle \xi \mapsto \{\nu : \texttt{buffer} \mid \mathsf{odd}(\nu)\}\rangle.$$

We discuss the results in turn. For the first example, `dsolve` takes roughly 0.8s. As expected, dsolve derives types for this example. For instance, the type for the main copying loop, `copy2`, is exactly the one given in Section 2 up to α-renaming.

For the second example, the bug is detected in 0.3s. while calculating the resources. In particular, consider the following part of the code:

```
 9  let rec copy buf_full buf_empty =
10    let drain = post (write outs buf_empty) in
11    if eof ins then
12      wait drain
13    else begin
14      let fill = post (read ins buf_empty) in
15        wait fill; wait drain; copy buf_empty buf_full
16    end
```

The tool detects an error at line 14: a resource which corresponds to the current instance of `buf_empty`, is accessed by two different tasks at the same time. This corresponds to a potential race condition, and it is, in fact, exactly the point where we introduced the bug.

For the third example, `dsolve` takes about 0.8s. Here, an error is detected in a more subtle way. The derived type of `copy` is:

$$\forall \mu_1.\texttt{buf_full} : \mathsf{ref}_{\mu_1}\,\texttt{buffer} \to \forall \mu_2.\texttt{buf_empty} : \mathsf{ref}_{\mu_2}\,\texttt{buffer}$$
$$\langle \mu_2 \mapsto \texttt{buffer} * \mu_1 \mapsto \{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}\rangle \to \mathsf{unit}\langle \ldots \rangle$$

In particular, in the initial call `copy buf2 buf1`, it must hold that `buf2` corresponds to any buffer, and `buf1` corresponds to a buffer with odd parity. To enforce this, dsolve introduces a subtyping constraint $\vdash \mu_1 \mapsto \{\nu : \texttt{buffer} \mid \rho\} \preceq \mu_1 \mapsto \{\nu : \texttt{buffer} \mid \rho'\}$, where ρ is the predicate that is derived for the content of the cell μ_1 at the moment when `copy` is actually called, and ρ' is the predicate from the function precondition, i.e., $\rho' = \mathrm{odd}(\nu)$. For ρ, dsolve derives the instantiation $\rho = \neg\mathrm{odd}(\nu)$. Therefore, the following subtyping constraint is asserted:

$$\vdash \mu_1 \mapsto \{\nu : \texttt{buffer} \mid \neg\mathrm{odd}(\nu)\} \preceq \mu_1 \mapsto \{\nu : \texttt{buffer} \mid \mathrm{odd}(\nu)\}$$

This constraint entails that for every ν, $\neg\mathrm{odd}(\nu)$ implies $\mathrm{odd}(\nu)$, which leads to a contradiction. Thus, dsolve detects a subtyping error, which points to the bug in the code.

5.2 Another asynchronous copying loop

The “Real World OCaml” book [20, Chapter 18] contains an example of an asynchronous copying loop in monadic style:

[3] Strictly speaking, `read` is a wrapper function, so it is not annotated with a type. Nevertheless, this is the type that it derives from its abstract implementation, `read_impl`.

$\texttt{Reader.read: Reader.t} \to$
$\quad \forall \mu, b \colon \mathsf{ref}_\mu\, \texttt{string}\langle \mu \mapsto \texttt{string}\rangle \to И\pi.\, \mathsf{promise}_\pi\, \texttt{result}\langle \mu \mapsto \texttt{string}\rangle,$
$\texttt{Writer.write:}$
$\quad \texttt{int} \to \texttt{Writer.t} \to \forall \mu, b \colon \mathsf{ref}_\mu\, \texttt{string}\langle \mu \mapsto \texttt{string}\rangle \to \mathsf{unit}\langle \mu \mapsto \texttt{string}\rangle,$
$\texttt{Writer.flushed: Writer.t} \to И\pi.\, \mathsf{promise}_\pi\, \mathsf{unit}\langle \mathsf{emp}\rangle.$

Figure 8 Types for asynchronous I/O functions in the Async library.

```
let rec copy_block buffer r w =
  Reader.read r buffer >>= function
  | ‘Eof -> return ()
  | ‘Ok bytes_read ->
    Writer.write w buffer ~len:bytes_read;
    Writer.flushed w >>= fun () -> copy_blocks buffer r w
```

where the functions `Reader.read`, `Writer.write` and `Writer.flushed` have the types given in Figure 8. One possible implementation of `Reader.read` is the following:

```
let read stream buffer = post (sync_read stream buffer)
```

where `sync_read` is typed as $\texttt{stream} \to \mathsf{ref}\,\texttt{buffer} \to \texttt{int}$, returning the number of bytes read. In practice, this function is implemented as an I/O primitive by the Async library, making use of operating system facilities for asynchronous I/O to ensure that this operation never blocks the execution of runnable tasks. The same holds for `Writer.write` and `Writer.flushed`.

By running `dsolve` on the example, we expect the following type for `copy_block`:

$$\forall \mu.b : \mathsf{ref}_\mu\, \texttt{string} \to r : \texttt{Reader.t} \to w : \texttt{Writer.t}\langle \mu \mapsto \texttt{string}\rangle \to$$
$$И\pi.\, \mathsf{unit}\langle \mathsf{Wait}(\pi, \mu \mapsto \texttt{string})\rangle$$

To be able to type this function, it needs to be rewritten in **post**/**wait** style. In this and all following examples, we use a specific transformation: In the Async and Lwt libraries, tasks are represented using an asynchronous monad with operators `return` and `bind`, the latter often written in infix form as `>>=`. A task is built by threading together the computations performed by the monad. For example, the following code reads some data from a Reader and, as soon as the reader is finished, transforms the data by applying the function `f`:

```
Reader.read stream >>= fun x -> return (f x)
```

This code can be translated to the post/wait style as follows:

```
post (let x = wait (Reader.read stream) in f x)
```

The idea is that the monadic value above corresponds to a single task to be posted, which evaluates each binding in turn. In general, a monad expression e_1 `>>=` e_2 `>>=` $\ldots$ `>>=` e_n can be translated to:

```
  post (let x_1 = wait e_1 in
        let x_2 = wait (e_2 x_1) in
        ...
        let x_n = wait (e_n x_{n-1}) in
        x_n)
```

The expression `return` e then translates to **post** e. Additionally, we use the "`return` rewriting law" `return` e_1 `>>=` $e_2 \equiv e_2\ e_1$ to simplify the expressions a bit further.

Running `dsolve` on the example takes about 0.1s, and derives the expected type for `copy_block`.

5.3 Coordination in a parallel SAT solver

The next example is a simplified version of an example from X10 [34]. It models the coordination between tasks in a parallel SAT solver. There are two worker tasks running in parallel and solving the same CNF instance. Each of the tasks works on its own state. A central coordinator keeps global state in the form of an updated CNF. The worker tasks can poll the coordinator for updates; this is implemented by the worker task returning POLL. The coordinator will then restart the worker with a newly-created task.

We use two predicates, `sat` and `equiv`. It holds that $\texttt{sat}(c)$ iff c is satisfiable. We introduce `res_ok cnf res` as an abbreviation for $(res = \texttt{SAT} \Rightarrow \texttt{sat}(cnf)) \wedge (res = \texttt{UNSAT} \Rightarrow \neg\texttt{sat}(cnf))$. The predicate `cnf_equiv cnf1 cnf2` holds if `cnf1` and `cnf2` are equivalent. Denote by $\texttt{cnf}_{\equiv c}$ the type $\{\nu : \texttt{cnf} \mid \texttt{cnf_equiv c } \nu\}$.

```
1  (* Interface of helper functions. *)
2  type cnf
3  type worker_result = SAT | UNSAT | POLL
4  val worker: c:cnf → ∀μ.ref_μ cnf⟨μ ↦ cnf_≡c⟩ → {ν : worker_result | res_ok c ν}⟨μ ↦ cnf_≡c⟩
5  val update: c:cnf → ∀μ.ref_μ cnf⟨μ ↦ cnf_≡c⟩ → cnf_≡c⟨μ ↦ cnf_≡c⟩
6  (* The example code *)
7  let parallel_SAT c =
8    let buffer1 = ref c and buffer2 = ref c in
9    let rec main c1 worker1 worker 2 =
10     if * then (* non-deterministic choice; in practice, use a select *)
11       match wait worker1 with
12       | SAT -> discard worker2; true
13       | UNSAT -> discard worker2; false
14       | POLL ->
15         let c2 = update c1 buffer1 in
16         let w = post (worker c2 buffer1) in
17         main c2 w worker2
18     else
19       ... (* same code, with roles switched *)
20   in main (post (worker c buffer1)) (post (worker c buffer2))
```

Here, **discard** can be seen as a variant of **wait** that just cancels a task. The annotations used in the example are given in the first part of the code, "Interface of helper functions".

For this example, we expect a type for `parallel_SAT` along the lines of

$$c : \texttt{cnf}\langle\texttt{emp}\rangle \to \{\nu : \texttt{bool} \mid \text{sat}(c) \Leftrightarrow \nu\}\langle\ldots\rangle.$$

Executing `dsolve` on this example takes roughly 9.8s, of which 8.7s are spent in solving subtyping constraints. The type derived for `parallel_SAT` is (after cleaning up some irrelevant refinements):

$$c : \texttt{cnf}\langle\texttt{emp}\rangle \to И\mu_1, \mu_2. \{\nu : \texttt{bool} \mid \nu = \text{sat}(c)\}\langle\mu_1 \mapsto \texttt{cnf}_{\equiv c} * \mu_2 \mapsto \texttt{cnf}_{\equiv c}\rangle$$

This type is clearly equivalent to the expected type.

5.4 The MirageOS FAT file system

Finally, we considered a version of the MirageOS [16] FAT file system code,[4] in which we wanted to check if our tool could detect any concurrency errors. Indeed, using ALS types, we discovered a concurrency bug with file writing commands: the implementation has a race condition with regard to the in-memory cached copy of the file allocation table.

For this, we split up the original file so that each module inside it resides in its own file. We consider the code that deals with the FAT and directory structure, which makes heavy use of concurrency, and treat all other modules as simple externals. Since the primary goal of the experiment was to check whether the code has concurrency errors, we do not provide any type annotations.

Running `dsolve` takes about 4.4s and detects a concurrency error: a resource is accessed even though it is still wrapped in a wait permission. Here is a simplified view of the relevant part of the code:

```
1
2  type state = { format: ...; mutable fat: fat_type }
3  ...
4  let update_directory_containing x path =
5    post (... let c = Fat_entry.follow_chain x.format x.fat ... in ...)
6  ...
7  let update x ... =
8    ...
9    update_directory_containing x path;
10   x.fat <- List.fold_left update_allocations x.fat fat_allocations
11   ...
```

In this example, `x.fat` has the reference type $\mathsf{ref}_\mu\,\texttt{fat_type}$. By inspecting the implementation of `update_directory_containing`, it is clear that this function needs to have (read) access to `x.fat`. Therefore, the type of e_1 :=`update_directory_containing x path` will be along the lines of $\Gamma; \mu \mapsto \texttt{fat_type} * \eta \vdash e_1 : \text{И}\pi, \Xi.\, \tau\langle \mathsf{Wait}(\pi, \mu \mapsto \texttt{fat_type} * \eta')\rangle$. Moreover, by inspection of e_2 :=`x.fat <- List.fold_left ... x.fat fat_allocations`, one notices that it needs to have access to memory cell μ, i.e., its type will be along the lines of $\Gamma; \mu \mapsto \texttt{fat_type} * \eta'' \vdash e_2 : \varphi$. But for $e_1\,;\,e_2$ to type, the postcondition of e_1 would have to match the precondition of e_2: In particular, in both, μ should have the same wait prefix. But this is clearly not the case: in the postcondition of e_1, μ is wrapped in a wait permission for π, while in the precondition of e_2, it outside all wait permissions.

By analyzing the code, one finds that this corresponds to an concurrency problem: The code in `update_directory_containing` runs in its own task that is never being waited for. Therefore, it can be arbitrarily delayed. But since it depends on the state of the FAT at the time of invocation to do its work, while the arbitrary delay may case the FAT data structure to change significantly before this function is actually run.

6 Limitations

A major limitation of ALS types is that it enforces a strict ownership discipline according to which data is owned by a single process and ownership can only be passed at task creation

[4] The code in question can be found on GitHub at `https://github.com/mirage/ocaml-fat/blob/9d7abc383ebd9874c2d909331e2fb3cc08d7304b/lib/fs.ml`.

or termination. This does not allow us to type programs that synchronize using shared variables. Consider the following program implementing a mutex:

```
let rec protected_critical_section mutex data =
  if !mutex then
    mutex := false;
    (* Code modifying the reference data, posting and waiting for tasks. *)
    mutex := true;
  else
    wait (post ()); (* yield *)
    protected_critical_section mutex data

let concurrent_updates mutex data =
  post (protected_critical_section mutex data);
  post (protected_critical_section mutex data)
```

The function `concurrent_updates` does not type check despite being perfectly safe: there is a race on the mutex and on the data protected by the mutex. Similarly, we do not support other synchronization primitives such as semaphores and mailboxes (and the implicit ownership transfer associated with them). One could extend the ALS type system with ideas from separation logic to handle more complex sharing.

Also, the type system cannot deal with functions that are all both higher-order and state-changing. For example, consider the function `List.iter`, which can be given as

```
let rec iter f l = match l with
  | [] -> ()
  | x::l -> f x; iter f l
```

As it turns out, there is no way to provide a sufficiently general type of `iter` that allows arbitrary heap transformations of f: There is no single type that encompasses `let acc = ref 0 in iter (fun x -> acc := x + !acc) l` and `iter print l` – they have very different footprints, which is not something that can be expressed in the type system. Since our examples do not require higher-order functions with effects, we type higher-order functions in such a way that the argument functions have empty pre- and post-condition.

Finally, we do not support OCaml's object-oriented features.

7 Related Work

7.1 Dependent types

This work fits into the wider framework of dependent types. Dependent and refinement types are a popular technique to statically reason about subtle invariants in programs. There is a wide range of levels of expressivity and decidability in the different kinds of dependent types. For instance, indexed types [38, 35, 36] allow adding an "index" to a type, which can be used, for instance, to describe bounds on the value of a numerical type. While the expressivity of this approach is limited, type inference and type checking can be completely automated. Similarly, the refinement types in [17] provide a way to reason about the state of a value, e.g., whether a file handle is able to perform a given operation, by providing a way to associate predicates with types.

On the other end of the expressivity scale are languages such as Agda [21], Coq [19] and Cayenne [1] in which types are expressions. For example, types in Agda can encode formulas in an intuitionistic higher-order logic, and type inference is undecidable.

We design our type system with automation in mind. We build up on liquid types [28], whose refinement types are based on predicates in some SMT-solvable logic. This allows reasonably automatic inference of subtle invariants. While low-level liquid types [29] provide some support for dealing with heap state, this is only for an imperative setting. In particular, only basic C-style data types are supported, and concurrency is not considered. Our work builds upon these foundations by extending liquid types with a way to handle state and concurrency in a functional setting.

7.2 Aliasing and concurrency

Another stream of directly related work are logics for reasoning about the heap and concurrency. Separation logic [27] is a logic that is specifically designed to deal with the aliasing problem when reasoning about heaps. It achieves this by introducing a separating conjunction operator *, where $\varphi_1 * \varphi_2$ means that φ_1 and φ_2 must hold on disjoint portions of the heap. Building on that, concurrent separation logic [22] introduces the notion of resource ownership to separation logic, and provides reasoning principles for passing resources between tasks. The resource passing scheme is more general than that of our type system; adapting it is left as future work.

Permissions have been used in separation logic to reason about programs with locks [9, 10] and programs with dynamic thread creation [6]. For instance, Gotsman et al. [9] introduce a Locked predicate that indicates that a task holds a lock on a given heap cell, which can be used to control whether certain operations such as unlocking may be performed. Dodds et al. [6] use permissions to extend rely-guarantee reasoning into a setting with dynamically generated threads by way of fork-join parallelism. Their approach is quite similar to our use of wait permissions: Upon forking a thread t, they generate a permission $\mathsf{Thread}(t, T)$, where T is a formula describing the postcondition of the thread t, namely the states that are assumed when the thread terminates. Upon waiting, these resources are transferred to the thread that joins t.

Our reason for not directly using CSL is twofold. Since we are using a functional language, the type system already gives us a lot of information that we would have to re-derive when performing proofs in CSL. Second, most work on the automatization of separation logic [5, 2, 11] focuses on the derivation of complex heap shapes. This is not a priority in our work, since in OCaml, instead of using pointer-based representations, complex data structures would be represented using inductively-defined data types.

Our type system is based on Alias Types [32]. They provide a type system that allows precise reasoning about heaps in the presence of aliasing. The key idea of alias types is to track resource constraints that describe the relation between pointers and the contents of heap cells. These resource constraints describe separate parts of the heap, and may either describe a unique heap cell (allowing strong updates), or a summary of an arbitrary number of heap cells. The Calculus of Capabilities [4] takes a similar approach. low-level liquid types [29] uses a similar approach to reason about heap state, and extends it with a "version control" mechanism to allow for temporary reasoning about a heap cell described by a summary using strong updates.

Another type system-based approach to handling mutable state is explored in Mezzo [24]. In Mezzo, the type of a variable is interpreted as a permission to access the heap. For instance, as explained in [25], after executing the assignment to x in `let x = (2, "foo") in ...`, a new permission is available: x@(`int`, `string`), meaning that using x, one may access a heap location containing a pair consisting of an `int` and a `string`. Without further annotation, these permissions are not duplicable, quite similar to our points-to facts. In certain situations,

e.g., the types of function arguments, permissions can be annotated as *consumed*, meaning that the corresponding heap cell is not accessible in the given form anymore, or *duplicable*, meaning the heap cell can be aliased without any issues (e.g., for an immutable heap cell). There are also additional features for explicit aliasing information and control. ALS and Mezzo address similar goals, but in two different ways: While both present a type system-based approach for handling mutable state and concurrency, ALS uses a straightforward extension of the OCaml type system, while Mezzo provides an entirely new typing approach. The result of this is that ALS's expressivity is somewhat limited, but requires little annotation and has powerful type inference, whereas Mezzo is very expressive, but requires more annotations and has only limited type inference.

7.3 Static analysis

Static analysis of asynchronous programming models have been studied in the model checking community [31, 12, 13]. These results focus on finite-state imperative programs without dynamic allocation. In contrast, our type system works with unbounded data domains and dynamically allocated heap data.

8 Conclusion

Asynchronous liquid separation types add the ability to reason about shared state in concurrent processes to liquid types, thereby increasing the expressivity of liquid types while retaining automated inference. In our preliminary experiments, we have shown that ALS types allow us to detect race conditions and prove user-specified data invariants for asynchronous code written in a functional style. It will be interesting to extend ALS to reason about common synchronization idioms in asynchronous code as well as to the full OCaml programming language.

References

1. Lennart Augustsson. Cayenne – a language with dependent types. In *Advanced Functional Programming*, pages 240–267. Springer, 1999.
2. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.
3. Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.
4. Karl Crary, David Walker, and J. Gregory Morrisett. Typed memory management in a calculus of capabilities. In *POPL 1999*, pages 262–275, 1999.
5. Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS 2006*, pages 287–302, 2006.
6. Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP 2009*, pages 363–377, 2009.
7. Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI 1991*, pages 268–277, 1991.
8. The Go programming language. `http://golang.org/`.
9. Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS 2007*, pages 19–37, 2007.
10. Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP 2008*, pages 353–367, 2008.

11 Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In *APLAS 2010*, pages 304–311, 2010.

12 R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *POPL*, pages 339–350, 2007.

13 Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV 2010*, 2010.

14 Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis. Asynchronous liquid separation types. `http://plv.mpi-sws.org/ALStypes/`. Full version.

15 Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI 1988*, pages 260–267, 1988.

16 Anil Madhavapeddy and David J. Scott. Unikernels: the rise of the virtual library operating system. *Commun. ACM*, 57(1):61–69, 2014.

17 Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ICFP 2003*, pages 213–225, 2003.

18 Nick Mathewson. Fast portable non-blocking network programming with Libevent. `http://http://www.wangafu.net/~nickm/libevent-book/`.

19 The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2012. Version 8.4.

20 Yaron Minsky, Anil Madhavapeddy, and Hickey Jason. *Real-World OCaml*. O'Reilly Media, 2013.

21 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

22 Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

23 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

24 François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. In *ICFP 2013*, pages 173–184. ACM, 2013.

25 Jonathan Protzenko. Introduction to Mezzo. Series of two blog posts, starting at `http://gallium.inria.fr/blog/introduction-to-mezzo`, Jan 2013.

26 Veselin Raychev, Martin T. Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *OOPSLA 2013*, pages 151–166, 2013.

27 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002*, pages 55–74, 2002.

28 Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *PLDI 2008*, 2008.

29 P.M. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL 2010*, 2010.

30 The Rust programming language. `http://www.rust-lang.org/`.

31 Koushik Sen and Mahesh Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV*, LNCS 4144, pages 300–314. Springer, 2006.

32 Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *ESOP 2000*, pages 366–381, 2000.

33 Jérôme Vouillon. Lwt: a cooperative thread library. In *ML 2008*, pages 3–12, 2008.

34 SatX10: A scalable plug & play parallel solver. `http://x10-lang.org/component/content/article/37-community/applications/207-satx10.html`.

35 H. Xi. Imperative programming with dependent types. In *LICS 2000*, 2000.

36 H. Xi. Dependent ML: An approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.

37 H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI 1998*, pages 249–257, 1998.

38 H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL 1999*, 1999.

The Eureka Programming Model for Speculative Task Parallelism

Shams Imam and Vivek Sarkar

Rice University, Houston, TX, USA, `{shams,vsarkar}@rice.edu`

Abstract

In this paper, we describe the Eureka Programming Model (EuPM) that simplifies the expression of speculative parallel tasks, and is especially well suited for parallel search and optimization applications. The focus of this work is to provide a clean semantics for, and efficiently support, such "eureka-style" computations (EuSCs) in general structured task parallel programming models. In EuSCs, a eureka event is a point in a program that announces that a result has been found. A eureka triggered by a speculative task can cause a group of related speculative tasks to become redundant, and enable them to be terminated at well-defined program points. Our approach provides a bound on the additional work done in redundant speculative tasks after such a eureka event occurs.

We identify various patterns that are supported by our eureka construct, which include search, optimization, convergence, and soft real-time deadlines. These different patterns of computations can also be safely combined or nested in the EuPM, along with regular task-parallel constructs, thereby enabling high degrees of composability and reusability. As demonstrated by our implementation, the EuPM can also be implemented efficiently. We use a cooperative runtime that uses delimited continuations to manage the termination of redundant tasks and their synchronization at join points. In contrast to current approaches, EuPM obviates the need for cumbersome manual refactoring by the programmer that may (for example) require the insertion of `if` checks and early `return` statements in every method in the call chain. Experimental results show that solutions using the EuPM simplify programmability, achieve performance comparable to hand-coded speculative task-based solutions and out-perform non-speculative task-based solutions.

1998 ACM Subject Classification D.1.3 [Programming Techniques] Concurrent Programming – Parallel Programming

Keywords and phrases Async-Finish Model, Delimited Continuations, Eureka Model, Parallel Programming, Speculative Parallelism, Task Cancellation, Task Termination

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.421

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.1.1.6`

1 Introduction

A wide range of problems, such as combinatorial optimization, constraint satisfaction, image matching, genetic sequence similarity, iterative optimization methods, can be reduced to tree or graph search problems [26, 1, 33]. A pattern common to such algorithms to solve these problems is a *eureka* event, a point in the program which announces that a result has been found. Such an event curtails computation time by avoiding further exploration of a solution space or by causing the successful termination of the entire computation. For example, in optimization problems, a eureka event declares (and updates) the currently best-known result and can prune the computation by causing the termination of specific tasks that cannot

© Shams Imam and Vivek Sarkar;
licensed under Creative Commons License CC-BY

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 421–444

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

provide a better result. On the other hand, in satisfiability problems, the first eureka event can trigger the termination of the entire computation as a proof of existence has been found.

With the advent of the multicore era, future growth in application performance will primarily come from increased parallelism. While many efforts have focused on programming models that expose increased amounts of deterministic parallelism, we believe that it is also important to explore new programming model directions for speculative parallelism. Eureka-Style Computations (EuSCs) include search and optimization problems that could benefit greatly from speculative parallelism. However, writing parallel programs is a non-trivial, bug-prone, and complex endeavor in general, and the addition of speculation to current models can further add to the complexity. There is, hence, a need for programming models that support simple specification of parallel EuSC algorithms, combined with efficient parallel implementations.

One of the challenges in EuSCs is the efficient termination of multiple active tasks once a solution is published. Current termination techniques include the use of (a) terminating processes and threads [37], (b) exceptions for control flow as used in Microsoft's Task Parallel Library [25] and Java thread interrupts in blocking calls [34], (c) function-scoped cancellation points in Cilk `abort` [15] and OpenMP 4.0 [31], and (d) manual cooperative termination tokens as in Intel Thread Building Blocks [27]. As explained in Section 3, all these solutions have their limitations and are inadequate for supporting parallel EuSCs with high productivity and performance.

In this paper, we introduce the Eureka Programming Model (EuPM), an explicitly task parallel programming model that simplifies the expression of parallel EuSCs. The EuPM builds on a structured task parallel programming model (summarized in Section 2). It works by exploiting parallelism opportunities in computations that are divided into several *speculative tasks*; these tasks are called *speculative* because their results may or may not be needed. Section 3 motivates our approach to terminating speculative tasks once a result is published. Our solution uses a simplified cooperative termination technique that only requires a single method call at each eureka point - a point in a program that announces that a result has been found. The EuPM is described in Section 4. It promotes out-of-order executions and the constructs in our EuPM are expressive enough to encode many parallel programming patterns common to EuSCs (Section 5). These different patterns can also be safely combined or nested, thereby enabling both composability and reusability (Section 6).

We have implemented the EuPM as a Java-based task parallel cooperative runtime that runs on a standard Java Virtual Machine, and it is summarized in Section 7. Our approach could be implemented for parallel C++ programs as well. The burden of performing code transformations to ensure that all redundant tasks are terminated cleanly at well-defined program points is assumed by the compiler and runtime. We evaluate the performance of search and optimization benchmarks, when using standard task-based solutions, hand-coded cooperative speculative task-based solutions, and solutions based on our EuPM. Experimental results (Section 8) show that the EuPM solutions can deliver significant performance and productivity improvements over standard task-based solutions. The EuPM abstraction achieves acceptable overheads with performance that is comparable to that of hand-coded speculative task-based solutions, while the EuPM solutions are simpler to write. Section 9 discusses related work, and we summarize our conclusions and identify opportunities for future work in Section 10.

In summary, the contributions of this paper are as follows:

- Introduction of the Eureka Programming Model (EuPM) to simplify the expression and management of speculative parallel tasks, which are especially important for parallel search and optimization applications.

- A manifestation of the EuPM as a standard Java API, with compiler support for the cooperative termination of avoidable tasks at well-defined program points.
- Identification of various patterns that are well-suited for the eureka construct, but hard to implement using current parallel programming models. These patterns include search, optimization, convergence, and soft real-time deadlines.
- An implementation of the EuPM in a cooperative runtime for task parallelism that uses delimited continuations.
- An empirical evaluation of the productivity and performance benefits of the EuPM implementation on various EuSC benchmarks.

2 Background and Motivating Example

In the task-parallel model, the application execution can be modeled as a directed acyclic graph, where nodes represent computational tasks and edges define the data dependences among them. A runtime system then efficiently schedules tasks whose dependences have been satisfied over the available processing units, usually implemented as worker threads. The management of actual threads and related thread pools is done by the runtime and is transparent to the tasks in the program.

2.1 Async-Finish Programming Model

The Async/Finish Model (AFM) is a structured variant of the task-parallel Fork/Join Model. In the AFM, a task can *fork* a group of child tasks. These child tasks can recursively fork additional tasks. All these descendant tasks can potentially run in parallel with each other. Further, a parent/ancestor task can selectively *join* on a subset of child/descendant tasks to wait for their completion.

Tasks are created at *fork* points, the statement `async` ⟨*stmt*⟩ causes the parent task to create a new child task to execute ⟨*stmt*⟩ (logically) in parallel with the parent task. An inner `async` is allowed to read and operate on a variable declared in an outer scope. The runtime is responsible for the scheduling of tasks created by `async`s. The `finish` construct represents a *join* operation. The task executing `finish` ⟨*stmt*⟩ has to wait for all transitively spawned child tasks inside ⟨*stmt*⟩ to terminate before it can proceed past the `finish` construct. All computations execute inside a global finish scope for the main program: the computation is allowed to terminate when all tasks nested inside the global finish terminate. This rule ensures that each executing task has a unique *Immediately Enclosing Finish* (IEF) [6, 40, 38].

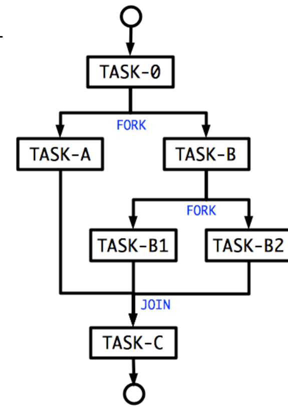
Listing 1 shows a sample program that uses `async` and `finish` constructs to preserve task dependences while exploiting available parallelism. Note that until all forked tasks (Task A, Task B, Task B1, and Task B2) reach the join point, Task C cannot be executed. The scopes of `async` and `finish` can span method boundaries that simplify parallelizing sequential programs. `async`s are inserted to wrap statements that can be executed in parallel and then these `async`s are wrapped inside `finish` blocks to ensure the parallel version produces the same result as the sequential version. `async-finish` style computations are guaranteed to be deadlock-free [6]. In addition, in the absence of data races, these programs are deterministic [38].

2.2 Parallel Search of 2D Array

We will consider a well-understood parallel programming example as a motivating example where speculative computation may be used for parallelization. We build on this example in

Listing 1 `async-finish` program using `async` and `finish` constructs for synchronization. (Listings use pseudo-code syntax.)

```
1  class AsyncFinishPrimer {
2    def main(args) {
3      finish // the global finish
4        println("Task 0") // Task-0
5        finish
6          async // Task-A
7            println("Task A")
8          async // Task-B
9            println("Task B")
10         async // Task-B1 created by Task-B
11           println("Task B1")
12         async // Task-B2 created by Task-B
13           println("Task B2")
14       // wait for tasks A, B, B1 and B2
15       println("Task C") // Task-C
16   } }
```



the rest of the paper to discuss various concepts and explain different EuSC variants. The example performs the parallel search of a 2D array to find the index of a particular item if it exists. The eureka event occurs when the item is found. The parallel version may expand (or generate) more states than a serial version. We are ready to tolerate such redundancy in the hope of a faster execution time in finding the result.

Listing 2 displays an implementation of such a parallel search using the `async` and `finish` constructs. Using the `atomicRefFactory()` method, the program initializes a `token` container to invalid indices at line 6. Tasks are spawned at line 9 and enclosed in the finish scope declared at line 7. The finish scope ensures that all spawned tasks complete before the result is returned at line 14. To overcome the overheads of tasking, a common strategy while working with arrays is to chunk the data. Each `async` processes a chunk of data using the `for` loop at line 10. The eureka event occurs at line 19 when the search successfully finds a match to the goal and updates the `token` atomic variable. Since we are interested in any result, we use `set` instead of a `compare-and-swap` operation. After the eureka event, the `procRow` method promptly returns.

Listing 2 `async-finish` parallel search.

```
1  class ParallelSearch {
2    def atomicRefFactory() {
3      return new AtomicRef([-1, -1])
4    }
5    def doWork(matrix, goal) {
6      val token = atomicRefFactory()
7      finish
8        for rowIndices in matrix.chunks()
9          async
10           for (r in rowIndices)
11             procRow(matrix(r), r, goal, token)
12     // return either [-1, -1] or
13     // valid index [i, j] matching goal
14     return token.get()
15   }
16   def procRow(array, r, goal, token) {
17     for (c in array.length())
18       if goal.match(array(c)) // eureka!!!
19         token.set([r, c])
20         return
21 } }
```

Listing 2 highlights a few inefficiencies, also common to other EuSCs, while using `async-finish` style parallelism. Firstly, there is the need to pass the task-specific `token` variable to each relevant method in the call chain to allow triggering the eureka event. Secondly, returning early from `procRow` doesn't terminate the task as the task can continue to process other iterations of the `for` loop at line 10. Next, there are other tasks executing concurrently which need to be terminated. If these tasks are long running, there can be a potentially large wait time before the finish scope ends, and the result is returned. Finally, as per `async-finish` semantics, all tasks (including those sitting in the work queue) will be executed even

after the result is known. We discuss existing solutions to these problems in terminating tasks and the drawbacks of such solutions in Section 3.

3 Task Termination Strategies

It is well-known that speculative computation can yield performance improvements over conventional approaches to parallel computing [4, 32]. The speculative tasks can be started eagerly before they are known to be required, for example, by spawning parallel tasks to search disjoint fragments of a data structure. Once a solution is found, other attempts at solving the problem may be avoided (in optimization) or terminated (in search). Eagerly terminating such tasks improves performance by minimizing the amount of unnecessary computations. One of the challenges in EuSCs is the efficient coordination of the termination of several related tasks. Supporting termination requires ensuring that a task can stop gracefully and leave the system in a state that is known to be valid.

One approach is terminating processes and threads [37]. This is not a scalable solution as the runtime then needs to spawn additional processes or worker threads to maintain the parallelism in the application. When worker threads are terminated repeatedly, the overheads of resource initialization cause the performance of the application to degrade. In addition, terminating a task abruptly might cause a computation to be interrupted asynchronously which can cause havoc in the programmer's understanding of the code's behavior. As in [24], we believe that it should not be possible to terminate a task in any execution state, but only at places where certain program invariants hold such that the execution may be interrupted safely. As a result, a preemptive approach is not desirable, and a cooperative approach where the task actively decides when to terminate is preferred.

One cooperative approach is the use of exceptions for control flow [25, 34]. Using exceptions allows the task to terminate with the runtime providing the exception handler to process that exception. It has the benefit that only specific program points defined by the programmer need to be edited to insert the `throw` clause; the bodies of callees need not be modified[1]. Use of exceptions to terminate tasks fail when users provide custom handlers that inadvertently catch the exception being thrown. This prevents the exception from reaching the handler provided by the runtime and thus interferes with the termination logic. A compiler can rewrite the exception handlers to immediately rethrow these special exceptions and prevent user code from interfering with termination logic [2]. However, this policy fails to work in the presence of *inaccessible* functions (whose source code is not directly available for modification). In addition, native support for throw code is comparatively inefficient even in the absence of filling the stack trace. The frequent use of exception handlers for control flow program execution logic is expensive and should be avoided.

Another cooperative approach is function-scoped cancellation points [15, 31]. For example, possible locations for cancellation points in Listing 2 would be at lines 9 to 11 which include the scope at which the `async` was declared. These work better as the compiler rewrites the code to support task termination; however, the limitation is that cancellation is not possible when the code is executing in a nested function call.

Another approach is manual cooperative termination via cancellation tokens or interrupt checking [27, 34]. Within long-running tasks, manually inserted periodic termination checks allow the task to determine if further work is avoidable (i.e. the task can be terminated). The granularity of checks controls the trade-off between the responsiveness of termination of

[1] Callee signatures may need to be modified to include the exception, however.

Listing 3 Parallel search with manual cooperative termination.

```
class ParallelCooperativeSearch {
   def atomicRefTokenFactory() {
      return new AtomicRefToken([-1, -1])
   }
   def doWork(matrix, goal) {
      val token = atomicRefTokenFactory()
      finish
         for rowIndices in matrix.chunks()
            async
               for r in rowIndices
                  procRow(matrix(r), r, goal, token)
                  // cooperative termination check
                  if token.isResolved()
                     return
      return token.get()
   }
   def procRow(array, r, goal, token) {
      for c in array.length()
         // cooperative termination check
         if token.isResolved()
            return
         if goal.match(array(c))
            token.set([r, c])
            return
   } }
```

tasks and the overhead of such check calls. Listing 3 displays the program from Listing 2 with support for manual cooperative termination. This approach is cumbersome to write as the programmer needs to manually transform all methods to support this style with an additional `token` parameter. It also requires `if` checks and early return statements (lines 13-14 and 20-21). Inserting such checks in the source code is awkward and impossible in the case of calls to inaccessible functions. If the computation includes inaccessible functions in the call stack, we need to wait for the body of each such function to complete before termination can be effected.

3.1 Delimited Continuation-based Cooperative Termination

Delimited Continuations (DeConts) were introduced by Felleisen in 1988 [14] where he referred to them as *prompts*. Other variants for DeConts include the `shift/reset` mechanism introduced by Danvy and Filinski [10]. Continuations represent the rest of a computation from any given point. They refer to the ability to *capture* the state of a computation at that point; the computation can later be *resumed* from that point by resuming the continuation. In contrast, DeConts represent the rest of the computation from a well-defined outer boundary, i.e. a sub-computation. DeConts work even in the presence of inaccessible functions in the call stack. When a computation is suspended, DeCont cause control to return to the caller of the boundary function irrespective of the functions (including inaccessible ones) that are in the call path. DeConts are hence a good choice when a limited part of the computation needs to be saved/restored [11]. Many mainstream languages offer support for various forms of DeConts, such as Boost coroutines in C++ [29], Kilim framework in Java [41], `shift/reset` in Scala [39], Ruby fibers [21], etc. DeConts are notorious for being hard to use and to understand by developers (as opposed to compilers and runtime systems). Hence, in a system that uses DeConts, an approach that does not expose a developer to DeConts is desirable.

Our approach to termination is cooperative and relies on the transparent use of DeConts. DeConts are required to let the control return to the runtime by safely unrolling the task's call stack but not the runtime worker's call stack, after which the runtime can perform

cleanup as if the task terminated or returned normally. The termination approach involves coordination between the code that requests termination (task that resolves a eureka) and the task that responds to termination (other executing tasks that have become redundant). Our approach guarantees that early termination of tasks can only occur at at well-defined program points: one of the check points or at points that resolve a eureka. If a task needs to be terminated, the call stack is unwound and control returned to the runtime via the use of DeConts. Such tasks are not resumed, so, unlike general DeConts, the state of the computation at that point need not be saved. Introducing a method call for the check introduces some overhead, so tasks can decide if they are *terminable* and how responsive they want to be to interruption. We expect a sweet spot, where a balance is reached between the overheads incurred by checks and the gains from early task termination. The optimal frequency of checking is application-specific and is determined by the developer. While the programmer does not need to hard-code exceptions and `if` checks on cancellation, our approach attempts to merge the benefits of using:

(a) **Exceptions:** There is no need for repeated `if` checks every time a method returns from a terminable method in the call chain. Only specific program points need to be edited to insert a method call. No exceptions are used for control flow, and the approach is unaffected by the presence of exception handlers.

(b) **Cancellation tokens:** It allows the programmer to determine the frequency of checks to terminate a task and determine responsiveness. However, our approach does not require changing the signature to add an additional parameter, the cancellation token is discovered implicitly by the task (further described in Section 4). Also, the body of inaccessible functions do not need to complete before termination of a terminable task can be effected.

The main limitation of our cooperative approach is that the programmer has to determine the frequency of the check calls and manually insert the termination check calls. These termination checks can also be inserted automatically by a compiler. Previous work by Feeley on *balanced polling* [13] and in the Jikes VM *yieldpoints* [44] provide a scheme to automatically insert these calls while minimizing overhead. Another limitation of our approach is that termination should not be triggered in a critical section that is implemented with programmer-defined locks. Acquiring a lock but failing to ensure that it is released can cause termination of the overall computation to be arbitrarily delayed; this issue plagues all the cooperative schemes discussed above. The issue can be circumvented by the use of managed resource control techniques that allow the runtime to track the lock(s) obtained by the user code while executing critical sections. These locks obtained by a task can then be released by the runtime when requested to do so. Such techniques include the use of custom virtual machines or language constructs to execute the critical section. For example, the Habanero programming model [23] includes such a language construct, called `isolated`, which would work with terminated tasks.

4 Programming with `Eurekas`

Parallel programming models ideally enable programmers to express parallel algorithms using abstractions that hide all but the relevant information to reduce complexity and to increase programmer productivity. Our goal is to define the Eureka Programming Model (EuPM) so that it can be used to write parallel programs for EuSCs more productively than current parallel programming models. In this section, we first introduce the `Eureka` construct that is used by speculative tasks to trigger eureka events. Next, we describe how parallelism is

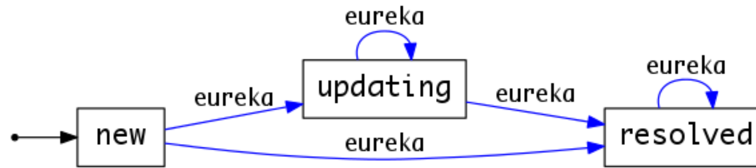


Figure 1 Life-cycle of `Eureka`. The states and transitions will become clearer when we introduce the different `Eureka` patterns in Section 5.

expressed via speculative tasks in the EuPM. We also explain how the termination of a single task, as well as a group of tasks, is supported in the EuPM.

4.1 `Eureka` Construct and API

A `Eureka` is a new construct that provides support for speculative parallelism in an `async-finish` setting. Once a `Eureka` construct has been *resolved* by reaching a stable value, it enables detection of a group of speculative tasks that can be terminated. It abstracts away implementation details, facilitates encapsulating any mutable state, and provides an API to allow tasks to concurrently notify eureka events as well as to query the state of the eureka object. Encapsulation simplifies data race avoidance while attending to concurrent eureka events triggered by speculative tasks.

As seen in Figure 1, a `Eureka` object has a well-defined life cycle; it can only transition between the states in response to eureka events. During its life cycle, a `Eureka` is in one of the following states:

(a) **new**: an instance of the `Eureka` has been created and initialized; however, it has not yet received any eureka events.

(b) **updating**: the `Eureka` has received at least one eureka event, and its internal state has not reached a stable value (e.g. computing minimum during optimization).

(c) **resolved**: the `Eureka` has reached a stable value; any subsequent eureka events may be ignored. Once a `Eureka` enters the resolved state, all speculative tasks that can trigger eureka events to update this `Eureka` become terminable and can be terminated.

We have developed an interface which supports basic behavior needed by tasks in EuSCs. This `Eureka` interface can be used to support a wide variety of patterns in EuSCs including search, optimization, and convergence. User's can also use this interface to build their custom implementations for `Eureka`s. The operations that can be performed on a `Eureka`, `eu`, are defined by the following interface:

(a) `offer`(`auxiliaryData`): Notifies `eu` that a eureka event has been triggered; additional information used to mutate the internal state of `eu` is available in `auxiliaryData`. This operation enables `eu` to transition to different states in its lifecycle (as shown in Figure 1). Whether or not the event resolves `eu`, it can cause the task invoking this operation to terminate at a well-defined program point.

(b) `check`(`auxiliaryData`): This operation allows a speculative task to check whether it has become terminable as, e.g., `eu` has been resolved. If the task has become terminable, a call to `check` will cause the task to be terminated. By accepting an argument, `check` enables the caller to pass additional values that can be used to determine whether to terminate a task.

(c) `isResolved`(): Allows a speculative task to query whether `eu` has been resolved. This method returns a boolean value and never causes a task to be terminated.

(d) **get**(): If eu has been resolved, it returns the resolved value. Otherwise, a transitory value of eu is returned. One is guaranteed to receive the resolved value if this operation is invoked outside the finish scope on which eu was registered (e.g. as we will see in the explanation of Listing 4 line-12).

4.2 Eureka Programming Model (EuPM)

The EuPM is an extension of the task-parallel async-finish model where speculative tasks are created using the async keyword. Through the hierarchical nature of async and finish blocks, we advocate a structured approach to parallel programming of EuSCs. The task executing a finish has to wait for all transitively spawned child tasks created inside the finish scope to terminate before it can proceed. A finish block can register on a Eureka, eu, with the following pseudocode syntax (the library API includes eu as a parameter to the finish API): finish(eu) ⟨*stmt*⟩. The finish construct simplifies the identification of the group of tasks that participate in a eureka-style synchronization on a particular Eureka instance.

All tasks having the same immediately enclosing finish (IEF) belong to the same group and inherit the registration on the Eureka instance, eu, from the finish scope. Finish scopes with different Eureka instance registrations can be nested allowing composability of different speculative computations. Similarly, multiple finish blocks can register on the same Eureka instance, eu, to represent that different speculative sub-computations are linked. When one of the speculative tasks resolves eu it makes other tasks from the same or different groups also registered on eu to become redundant and terminable. If none of the tasks trigger a eureka event that resolves the registered eu, the computation completes normally when all tasks inside each finish scope complete.

The EuPM specific operations that a task, T, can perform on a Eureka, eu, are defined as follows:

(a) **new**: Task T can create a new instance of the Eureka construct, eu, and obtain a handle to it. The reference eu can now be used to register on new finish scopes. The creator task can pass the reference of eu to other tasks.

(b) **registration**: eu can be explicitly registered on a finish scope. Note that the task that created eu cannot register on eu. A newly spawned task, T, implicitly registers on eu only if the IEF of T was explicitly registered on eu. Currently, we do not provide a mechanism for an async task to explicitly register on eu.

(c) **offer**: This method retrieves the Eureka instance, eu, that the task is registered on and invokes the eu.offer() method to notify eu of a eureka event. This simplifies the computation body of T where method calls do not need to add an extra parameter to pass eu down the call chain. Invoking this method can cause the task to terminate (depending on the implementation of eu).

(d) **check**: A task indirectly performs a check on eu by invoking the static check method. Like the previous operation, it retrieves eu to make the call eu.check(). Invoking this method can cause the task to terminate (depending on the terminating logic of eu), so programmers need to ensure that side-effects introduced by T are in a consistent state at the program point where this method was invoked.

With the EuPM, a programmer can focus on writing operational code explicitly specifying potentially parallel operations, leaving the underlying details of parallel execution and termination detection to the runtime system. The EuPM places no requirements on the use of a shared-memory infrastructure. Like the async-finish model, the EuPM presented in this paper is also applicable to a distributed environment. One of the key features of a

system that supports EuSCs is the efficiency with which the eureka events are triggered. The EuPM provides the abstraction, static `offer` method, that simplifies how eureka events are triggered in tasks. Invalid calls to `check`/`offer` from a task not executing in a EuSC (i.e. `finish` scope not registered on a `Eureka`) results in a runtime error.

Another feature is the efficiency with which the state of the program can be updated after the result has been found. The EuPM needs to provide a means to easily detect tasks that need to be terminated and a mechanism to guarantee effective termination of terminable tasks. Once a `Eureka` instance, `eu`, moves to the resolved state, all incomplete tasks belonging to a finish scope registered on `eu` become terminable. Any ready tasks belonging to the *resolved* finish scope can be terminated by removing them from the work queue – each task is assumed to contain an implicit `check` call at the start of the task execution. Redundant executing tasks are terminated at program points where the `check` method is invoked. This allows tasks to terminate cooperatively in a programmer controlled manner and, more importantly, simplifies reasoning about the correctness of the speculatively parallel program.

Listing 4 displays the parallel search program of Listing 2 using `async` and `finish` constructs in the EuPM. We create the `SearchEureka` instance, `eu`, inside the factory method `eurekaFactory`. This instance, `eu`, is registered by the `finish` scope defined on line 7. Hence, all `async` tasks launched at line 9 are automatically registered on `eu` and belong to the same group. The tasks trigger the eureka event by invoking the `offer` method at line 18. There is no need for an explicit `return` statement in `procRow`, as `offer` on a `SearchEureka` causes the task to terminate. To enable cooperative termination, there are also calls to `check` (line 16) to check the state of the registered eureka implicitly. When `eu` has been resolved, `check` will cause the terminable tasks to terminate. Eventually, all tasks inside the `finish` block at line 7 will complete execution or be terminated, and the computation will proceed to line 12 and the result will be returned. Note that, like Listing 3, the final answer in this example is nondeterministic, but there are no data races involved. It should be noted that this program (19 lines) required fewer lines of code than the equivalent program in Listing 3 (25 lines). In addition, the code in Listing 3 is more complicated and error-prone than the code in Listing 4. As we will see in Section 5, we will build on this example to explore various EuSC patterns with minimal change in the kernel code (lines 5 to 19 of Listing 4).

Listing 4 Parallel search using the Eureka model.

```
1  class ParallelEurekaSearch {
2     def eurekaFactory() {
3        return new SearchEureka([-1, -1])
4     }
5     def doWork(matrix, goal) {
6        val eu = eurekaFactory()
7        finish (eu) // eureka registration
8           for rowIndices in matrix.chunks()
9              async
10                for r in rowIndices
11                   procRow(matrix(r), r, goal)
12       return eu.get()
13    }
14    def procRow(array, r, goal) {
15       for c in array.length()
16          check([r, c]) // coop. term. check
17          if goal.match(array(c))
18             offer([r, c]) // trigger eureka
19    } }
```

5 Parallel Patterns and Eureka Variants

In this section, we describe frequently occurring patterns that arise in EuSCs and how they can be solved using the EuPM. These patterns include computations that produce both deterministic and non-deterministic results.

5.1 Parallel Search

Search is a well-known pattern in EuSCs. It is a non-deterministic computation in the sense that if the goal is present at multiple locations in the data structure being searched, any of those locations is an acceptable result. Searching disjoint partitions of a data structure can be done in parallel though it may considerably increase the amount of work that the algorithm performs. Such parallelism is speculative since more than one partition may contain a solution. Once the result is discovered, all parallel searching entities should ideally be terminated as quickly as possible to minimize doing redundant work. With respect to the EuPM, this means that the first eureka event triggered by a task will resolve the `Eureka` instance, `eu`, registered by the task. Hence, a `SearchEureka` construct is designed to be resolved by the first eureka event it processes, and it promptly terminates the task that triggered the event. Any subsequent calls to `check` or `offer` by other tasks registered on `eu` result in those tasks being terminated.

The same concept can be used for termination detection in the `finish` statement with regards to exception semantics. If any `async` throws an exception, then it can resolve an implicit `SearchEureka` registered by the `finish` scope. All other `async`s belonging to the same IEF can then be terminated at their next `check`/`offer` checkpoint. The IEF can then rethrow the exception thrown by the `async`. This offers an alternate strategy to the *MultiException* scheme [6] where a collection of all exceptions thrown by all `async`'s in the IEF is rethrown.

5.2 Count Eureka

Another variant of a parallel search is where we wish to know the first K results that match a query. This pattern is inspired by the `ParallelTry` command in Mathematica 7 [47]. In this pattern, we wish to terminate the computation when at least K of the asynchronous computations have completed successfully. Any evaluations still underway after K results have been received are avoidable and should be terminated. Like the `SearchEureka` pattern, the `CountEureka` pattern produces non-deterministic results as the results received are dependent on the scheduling of parallel tasks and the arrival of concurrent eureka events.

The program from Listing 4 can be modified to use the `CountEureka` construct by changing the factory method. A `CountEureka` is initialized with a count K and is resolved after exactly K eureka events have been triggered. Once resolved, any calls to `offer` and `check` cause the calling task to be terminated. A call to `CountEureka.get()` returns a list of values of maximum length K instead of a single value. If none of the tasks triggered a eureka, then an empty list is returned. In general, a `SearchEureka` can be viewed as a `CountEureka` with a count of 1.

5.3 N-Version Eureka

N-Version Programming [7] uses software redundancy to achieve fault-tolerance. In N-Version Programming, there are multiple functionally equivalent implementations of the same specification. These implementations can run be independently in parallel to compute results; the results are notified using eureka events. Using a decision algorithm, such as when any N (≥ 2) agree on their results, the eureka is resolved. The agreed upon value is accepted as the result matching the specification, and other computations are terminated. This pattern also produces non-deterministic results as the final result is dependent on the scheduling of parallel tasks and the arrival of concurrent eureka events from the independent implementations.

5.4 Optimization Eureka

Many problems from artificial intelligence can be defined as combinatorial optimization problems. For example, Branch and Bound (BnB) is a widely used tool for solving large-scale NP-hard combinatorial optimization problems [8]. A BnB algorithm searches the complete space of solutions for a given problem for the best solution. Subproblems are derived from the originally given problem through the addition of new constraints. An objective function computes the lower/upper bounds for each subproblem. The upper bound is the worst value for the potential optimal solution; the lower bound is the best value. The entire tree maintains a global upper bound (GUB): this is the best upper bound of all nodes. Nodes with a lower bound higher than the GUB are eliminated from the tree because branching these sub-problems will not lead to the optimal solution. In many practical cases, the amount of pruning that occurs in this type of BnB algorithm can be very significant.

In parallel implementations, pruning the branches of the search tree may lead to terminating existing computations. The structure of the BnB search requires the ability to terminate individual subtrees of the search tree [35]. A BnB version of our array search example is where we are interested in finding the lowest index of the goal item if it exists in the array. We can achieve this by modifying the factory method in Listing 4 to return a `MinimaEureka` instance. In our EuPM, the GUB is available in the `MinimaEureka` instance, `eu`, that a speculative task is registered on and can be retrieved by a call to `eu.get()`. Calls to `check` and `offer` pass the current known upper bound or solution, respectively, as the argument. If the argument in the `offer` call is lower than the GUB, the GUB is updated in the `MinimaEureka` instance, otherwise the current task is terminated. Similarly, calls to `check` terminate a task if the argument is larger than the currently known GUB in `eu`.

5.5 Soft Deadlines

For soft real-time systems [5] the goal is to meet a certain subset of deadlines to optimize some application-specific criteria. If a soft real-time task takes longer than the allotted time since its creation to complete, then it needs to be terminated with its latest results. Another similar notion is that of engines that abstract the notion of timed preemption [20]. An engine is run by giving it a quantity of abstract time units that measure computation. If the engine completes its computation before running out of units, it returns the result of its computation and the quantity of remaining units. If it runs out of units, the computation is terminated. Unlike Haynes' original notion of engines, nesting of engines is allowed in our model thus allowing time units to be divided among parallel sub-tasks if required.

In our soft deadline version of the array search example, the deadlines could be overall execution time (soft real-time) or number of comparison operations performed (abstract time units). The eureka version of these programs helps the system by releasing the resources of the tasks which have already missed their deadlines and allocating more resources to the other tasks which can still potentially meet their deadlines. We support both kinds of `Eureka`s in the form of `TimerEureka` and `EngineEureka`, respectively. A `TimerEureka`, `eu` is resolved when either the first eureka event is triggered or the computation runs out of time since the creation of `eu`. An `EngineEureka`, `eu` is resolved when either the first eureka event is triggered or the computation runs out of time units (measured by the sum total of the arguments to `check`). These `Eureka` instances can trigger the termination of a group of tasks without an explicit `offer` from a task. Note that since tasks only get cancelled when they invoke `check`, tasks can run for much longer than the allotted time unless the user is careful with the calls to `check`. This is consistent with our philosophy of allowing the programmer to determine responsiveness (Section 3).

5.6 Convergence Iterations

Iterative methods [17] refer to a wide range of techniques that use successive approximations to obtain more accurate solutions to a set of equations at each step. Examples of iterative methods include the Jacobi method, Gauss-Seidel method, and the Successive over-relaxation method. An iterative method is called *convergent* if the corresponding sequence converges for given initial approximations. Speculatively parallelizing an iterative algorithm results in creating tasks for computations of *future* iterations.

Listing 5 displays an example which computes r using the equation: $x_{i+1} = f(x_i)$, $y_{i+1} = g(y_i)$, $r_i = h(x_i, y_i)$ and converges when successive values of r become *close*. The parallel version launches `maxIters` iterations ahead of time (line 22) and parallelizes the computation to respect the dependences (using the `await` clause). The `await(T1, ..., TN)` clause in a task causes it to be suspended from execution until the execution of tasks `T1, ..., TN` has completed.

Listing 5 Example of an iterative method using the Eureka model.

```
1  class ParallelEurekaConvergence {
2    def eurekaFactory(initVal, tolerance) {
3      val pred = (a, b) -> {
4        Math.abs(decimalDiff(a, b)) <= tolerance
5      }
6      return new ConvergenceEureka(initVal, pred)
7    }
8    def doWork() {
9      val maxIters = 100
10     val initVal = INFINITY
11     val tolerance = 1e-4
12     val eu = eurekaFactory(initVal, tolerance)
13     finish (eu) {
14       // arrays to store task handles
15       val xs = newArray(maxIters + 1)
16       val ys = newArray(maxIters + 1)
17       val rs = newArray(maxIters + 1)
18       xs[0] = async { return xInit }
19       ys[0] = async { return yInit }
20       rs[0] = async { return initVal }

22       for (i in 1 to maxIters) {
23         xs[i] = async {
24           await(xs[i - 1]) // dependence
25           check()
26           return f(xs[i - 1])
27         }
28         ys[i] = async {
29           await(ys[i - 1]) // dependence
30           check()
31           return g(ys[i - 1])
32         }
33         rs[i] = async {
34           check()
35           await(xs[i], ys[i], rs[i - 1])
36           val iterRes = h(xs[i], ys[i])
37           // converge if rs[i] and rs[i-1] close
38           offer(iterRes)
39           return iterRes
40     } } }
41     return eu.get()
42 } }
```

The `await` clause in line 35 ensures that values of r are offered to `eu` in the expected order. Once convergence is reached (i.e. when true is returned by the call to `predicate` inside `eu` with the current and new value) `eu` is resolved and all tasks spawned by the computation need to be terminated. Note that this includes tasks that may be transitively suspended on `await` clauses as each await task is assumed to contain an implicit `check` call at the start of task execution (Section 4.2). Calls to `check` in the `async`s and possibly inside methods `f`, `g`, and `h` ensure executing tasks can be terminated early.

6 Reusability and Composability of Eureka Components

Abstractions and productivity are among the most important requirements for programming models. Further, it is important for the abstractions to be as orthogonal as possible, so as to aid composability and reusability. Constructing reusable components aids in programmer productivity by simplifying building of larger systems from relatively simpler parts. While current approaches to support EuSCs lack composability and reusability in general, we show in this section that all the different styles of eureka computations presented in Section 5 can be safely combined or nested thereby enabling general composability and reusability.

Listing 6 Example of a parallel search on two elements of an 2-D array using the Eureka model.

```
class ParallelEurekaDoubleSearch {
   def eurekaFactory() {
      val initValue = [-1, -1]
      return new SearchEureka(initValue)
   }
   def doWork(matrix, goal1, goal2) {
      val eu1 = eurekaFactory()
      val eu2 = eurekaFactory()
      val eu = eurekaComposition(AND, eu1, eu2)
      finish (eu) // eureka registration
         for rowIndices in matrix.chunks()
            async
               for r in rowIndices
                  procRow(matrix(r), r, goal1, goal2)
      // eu.get() returns pair of values
      return eu.get()
   }
   def procRow(array, r, g1, g2) {
      for c in array.length()
         // pair of values for eu1 and eu2
         val checkArg = [[r, c], [r, c]]
         // cooperative termination check
         check(checkArg)
         val loopElem = array(c)
         val res1 = g1.match(loopElem) ? [r, c] : null
         val res2 = g2.match(loopElem) ? [r, c] : null
         // pair of values for eu1 and eu2
         val foundIdx = [res1, res2]
         // possible eureka event
         offer(foundIdx)
   } }
```

6.1 Composability by Component Composition

Component composition involves the systematic combining of independent components to form useful components. Such incremental aggregation of existing components yields further components. This method is scalable as code replication is avoided while implementing new functionality. In the EuPM, independent `Eureka`s can be combined to form new `Eureka`s. The constituent `Eureka`s are used as encapsulated black-box components and are accessed solely through their exposed interfaces (the `check` and `offer` operations).

We support basic logical conjunctive and disjunctive binary composition semantics for `Eureka`s. Calls to `offer` and `check` need to be passed a pair of values, one for each component `Eureka`. The conjunctive composition of `Eureka`s is considered resolved only when both the constituent `Eureka`s are resolved. A disjunctive composition of `Eureka`s is considered resolved when either of the constituent `Eureka`s is resolved. Since the state of a `Eureka` instance evolves monotonically (once resolved a `Eureka` always remains resolved), the binary composed `Eureka` also preserves monotonicity, i.e., resolution of a `Eureka` is a stable property.

Listing 6 shows an example with a conjunctive eureka. We extend the example from Listing 4 to search for two target items in parallel and report a success only when both items are found. The example creates two search `Eureka`s and then creates a conjunctive eureka, `eu`, at line 9. This `eu` is used to register on the finish scope and launch the parallel search tasks on individual rows. Each call to `check` and `offer` now passes a pair of values (lines 23 and 30). The internal implementation delegates the individual values from the pairs to the component `Eureka`s. These individual values may end up resolving only one of the component `Eureka`s. The overall conjunctive eureka, `eu`, is resolved when both component `Eureka`s are resolved, possibly from different calls to `offer` in different tasks. Once `eu` has been resolved, calls to `check` (line 23) will result in the task being terminated.

6.2 Reusability by leveraging Functional Decomposition

Function decomposition refers to the process of splitting a computation into multiple fragments. These fragments are invoked as helper functions and their results combined to produce the overall result. Alternatively, it can also be viewed as the functional composition of existing functions so that the results of these function calls are used to evaluate a larger computation. Reuse is achieved by building functions on top of existing functions. Such uses of functions for individual computation fragments simplify maintainability and avoid code duplication.

In the EuPM, we enable opportunities for functional decomposition by allowing the nesting of EuSCs with `finish` and `async` statements. This nesting can be arbitrarily deep and allows exposing nested fork-join parallelism opportunities in distinct EuSCs. Nesting of `finish` blocks registered on `Eureka` instances is allowed and enables composability of different speculative computations. Each `finish` scope registered on the same `Eureka` instance forms a single group. When different EuSCs are nested, calls to `offer` are delegated to the registered `Eureka` on the IEF of the currently executing task. However, calls to `check` are recursively delegated to registered `Eureka` instances up the nested `finish` scope hierarchy. This allows the innermost EuSC to continue to work as before, but tasks may be aborted if `Eureka`s up the hierarchy have been resolved by other parallel tasks. Thus, nesting EuSCs causes a tree hierarchy of `Eureka`s to become linked whereby resolving a `Eureka` up the hierarchy causes computations lower in the tree to be terminated.

This nesting mechanism is explained in Listing 7 which shows an example to do a search in a multidimensional array. The solution reuses the `doWork` function from Listing 4 and is thus performing functional composition. Nested EuSCs are also created at line 14 where an individual `Eureka` instance is created for every row (line 13) in the leading dimension of the array in the recursive call. When the `Eureka`, `eu`, is resolved for a dimension `N1`, it causes all nested eureka computations processing dimension `N2` (where `N2` $<$ `N1`) to be treated as resolved. As a result, subsequent calls to `check` (at line 17) in the redundant tasks of the nested computations will cause the tasks to terminate. After returning from a call to a nested eureka computation, the result is either returned immediately (for the base case at line 11) or processed further (line 18). Further processing involves updating the result index with the value for the current dimension before offering the result via `offer` (line 20).

7 Implementation

Despite any productivity promises, a parallel programming model must be implementable in an efficient and scalable fashion for it to be accepted by programmers. Our implementation of the EuPM is an extension of a Java-based task parallel runtime [23] that supports cooperative scheduling of `async-finish` style computations, though our ideas can also be implemented in other languages including C/C++. Figure 2 highlights the features and responsibilities of the task-parallel Eureka runtime in our model. These include implementation of efficient eureka variants; management and scheduling of the speculative tasks; classification of tasks into eureka groups; termination of redundant tasks; and synchronization and coordination of tasks

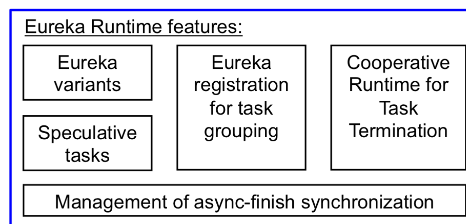


Figure 2 Features supported by a Eureka task parallel runtime.

Listing 7 Example of parallel search on a multidimensional array using function decomposition and nested eureka computations in the Eureka model.

```
1  class ParallelEurekaArraySearch {
2     def eurekaFactory(dim) {
3        val initValue = array(dim).fill(-1)
4        return new SearchEureka(initValue)
5     }
6     def doWork(matrix, dim, goal) {
7        if (dim < 2)
8           throw IllegalArgumentException("Invalid dimension: " + dim)
9        else if dim == 2
10          // reuse by call to existing eureka computation  Listing 4
11          return ParallelEurekaSearch.doWork(matrix, goal)
12       else
13          val eu = eurekaFactory(dim)
14          finish (eu) // nested eureka registration from recursive calls
15             for i in matrix.length()
16                async
17                   check(array(dim-1).fill(-1).insert(i)) // termination check
18                   val resIndices = doWork(matrix(i), dim-1, goal)
19                   if isIndexNonNegative(resIndices)
20                      offer(copy(resIndices).insert(i))
21          return eu.get()
22 } }
```

inside `finish` blocks. The challenges in the implementation of the EuPM involve effective load balancing of tasks, terminating tasks in a group efficiently, and supporting `Eureka`s in a scalable manner.

Our implementation supports all the `Eureka` variants described in Section 5 based on the API defined in Section 4.1. The key challenge is to support the synchronization by keeping each thread busy without any blocking. We rely on Java's support for atomic variables to implement the `Eureka`s and detect when a `Eureka` has been resolved. In particular, we rely heavily on the use of *compare-and-swap* operations, on `AtomicBoolean` and `AtomicLong` instances. This ensures thread safety and avoids data races in the concurrent calls to `check` and `offer` on the `Eureka`.

Our runtime uses the help-first policy [18] and maintains an independent stack for each task. Hence, any worker thread may execute a task, and we can use either work-stealing or work-sharing scheduling policies in our runtime. Since subproblems are generated and consumed dynamically, we rely on the load balancing algorithm provided by Java's `ForkJoinPool`. The `ForkJoinPool` is an optimized thread pool, which employs a work-stealing mechanism to efficiently distribute submitted tasks among its pool threads. Figure 3 displays how the runtime uses work-stealing threads to schedule tasks. Each task also maintains a reference to its IEF and inherits the `Eureka` registration from its IEF. This `Eureka` instance is stored in the IEF and used as the recipient while delegating calls to `check` and `offer`. Thus the tasks registered with the same `Eureka` instance are implicitly grouped together and become terminable when the `Eureka` is resolved. The tasks are eagerly terminated when there is a call to `check` or `offer`. Tasks executing inside a `finish` scope not registered with a `Eureka` (e.g. `FIN-0`) execute as regular `async-finish` style tasks without support for early termination.

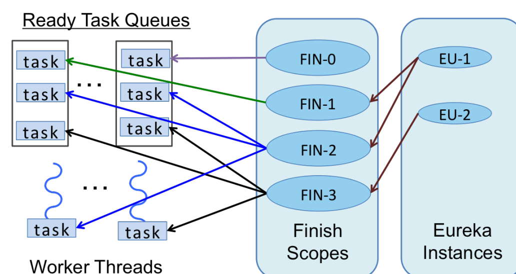


Figure 3 Execution of parallel Eureka tasks in a work-stealing environment.

As mentioned in Section 3, we rely on a cooperative task termination technique. When a task needs to be terminated at one of the check points, the call stack is unwound and control returned to the runtime via the use of Delimited Continuations (DeConts) [14]. Our implementation conforms to the constraints imposed by a standard Java Virtual Machine (JVM). In particular, standard JVMs do not provide support for DeConts or for storing and restoring the stack that we need to support cooperative termination. We have built a cooperative runtime that schedules tasks in the presence of end-of-finish synchronization constraints without blocking underlying worker threads using the strategy described in [22]. The DeConts created are thread independent and can be resumed on any worker thread. This strategy is known to provide a more scalable solution than other schemes that use thread-blocking operations [22].

We use an extended version of the open source bytecode weaver provided by the Kilim framework [41] to support DeConts. The Kilim bytecode weaver works by transforming the code of methods which can terminate. It recognizes such methods by the presence of a `SuspendableException` exception in the method signature. It is important to note that no actual exceptions are thrown or caught which minimizes the overhead of capturing and resuming continuations. Instead, the transformation performed is similar to a continuation passing style transformation, except that only methods that can suspend are transformed. The weaver also allocates custom state objects to store local variables to enable restoring the computation. We extended Kilim to enable execution of parallel tasks run by worker threads in the `ForkJoinPool`. Then we added support for terminated tasks. Such tasks are never resumed, so unlike general DeConts the state of the computation at that point need not be stored. This avoids additional memory allocation and garbage collection overheads while suspending the DeCont.

Next, we extended the classical `async-finish` constructs with support for the proposed `finish` and `eureka` constructs used in EuSCs. We provide support for EuPM `finish` constructs that register on a `Eureka`, `eu`, and each time a task is spawned from the finish scope, the task is also registered on `eu`. Nested finish scopes can register on different `Eureka` instances as each finish scope maintains its own `Eureka`. This enables composability of different speculative computations. Static helper methods, such as `check` and `offer`, then retrieve the `Eureka`, `eu`, registered with the currently executing task from its IEF before delegating the call on `eu`. Whenever a eureka is resolved, the finish scope, `f`, is notified, and all tasks whose IEF is `f` and that are in the work queue are terminated immediately. In-flight executing tasks belonging to `f` are terminated at `check` or `offer` points as termination is cooperative. Terminating executing tasks is done by suspending the current DeCont and flagging it as terminated, so that the runtime can perform cleanup operations and schedule other tasks for execution. Once all of these tasks have been successfully terminated, the end-of-finish point for `f` is resumed.

8 Experimental Results

The benchmarks were run on individual nodes in an IBM POWER7 compute cluster. Each node contains 256GB of RAM and four eight-core IBM POWER7 processors running at 3.8GHz each. The software stack includes IBM Java SDK Version 7, and our implementation of the cooperative runtime version 0.1.2. We configured each benchmark to run using 32 worker threads and run for thirty iterations in six separate JVM invocations using the same

Table 1 Configurations of the benchmarks: SLS on a $1,000 \times 2,500,000$ array, with the result located at index (350, 875000). UTS using the UTS T1 configuration, a geometric tree with a branching factor of 4 and a maximum height of 12; graph of size 164 million. SUD on a 25×25 board with 196 unsolved entries. NQK computes first 250 thousand solutions on board size of 15×15. UTP tree using the UTS T3 configuration, a binomial tree with a maximum height of 11; 700 goal nodes; and graph of size 13 million nodes. FLP on a 64×64 grid with 14 cells. TSP on 24 cities. KMC with 3 million points partitioned into 15 clusters. J2D using an array of size $5,000 \times 5,000$ with a block size of $1,000$. DLS on a $1,000 \times 2,500,000$ array, with the results located at index (100, 250000) and (350, 875000). CS on a $20 \times 20 \times 60 \times 15,000$ array, with the result located at index (8, 8, 24, 6000).

Benchmark Name	Acronym	Source (Eureka Pattern)	Description
Single Linear Search	SLS	Authors (Section 5.1)	Search for a single item in a 2D array.
Unbalanced Tree Search	UTS	UTS [30], (Section 5.1)	Search for a goal node in UTS trees which represent characteristics of various parallel unbalanced search applications.
Sudoku	SUD	Authors (Section 5.1)	Solving a Sudoku puzzle by exploring a game tree.
NQueens first K solutions	NQK	Authors (Section 5.2)	Find first K solutions to placing N queens on a chessboard such that no queen can attack any other.
UT Shortest Path	UTP	Authors (Section 5.4)	Trees from the UTS benchmark, adds edge weights to find shortest path to any goal node.
BOTS Floorplan	FLP	BOTS [12], (Section 5.4)	Compute the optimal floorplan distribution of a number of cells using branch and bound technique.
Traveling Salesman Problem	TSP	R. Wiener [46], (Section 5.4)	Solved using a branch and bound algorithm.
Jacobi 2D	J2D	Authors (Section 5.6 style)	Stencil computation with iterative convergence.
K-Means Clustering	KMC	Authors (Section 5.6 style)	An iterative refinement technique which converges to a local optimum.
Double Linear Search	DLS	Authors (Section 6.1)	Search for two items in a 2D array.
Composite Search	CS	Authors (Section 6.2)	Search for a single item in a multi-dimensional array.

JVM configuration flags[2]. The arithmetic mean of the best fifty execution times (from the hundred and eighty iterations) are reported. Using the best execution time allows us to minimize the effects of JVM warm up, just-in-time compilation, and garbage collection.

Speculative Execution Benchmarks: The benchmarks are described in Table 1. The benchmarks include some of our motivating examples, search benchmarks, game puzzles, greedy algorithms, branch and bound algorithms, and a stencil computation. We present empirical evaluation of our implementation of the EuPM (`EU`) relative to variants that: (a) provide no support for early termination of `async-finish` tasks (`AF`); (b) use function-scoped cancellation points for termination of speculative `async-finish` tasks[3] (`FS`); (c) use exceptions for termination of speculative `async-finish` tasks[4] (`EX`); and (d) use `if` checks and `return` statements via cancellation tokens speculative `async-finish` tasks (`CT`). The results for execution time and productivity metrics are described below.

Execution Times Comparison: We compare the performance of the different `Eureka` patterns in the benchmarks. The comparison with the `AF` versions shows that most of these benchmarks benefit from speculation. In fact, in some of the benchmarks (e.g. SUD, TSP) the non-speculative variant does not complete execution. In other benchmarks, e.g. SLS, UTS, NQK, FLP, the non-speculative versions perform higher numbers of abstract operations (e.g. comparisons, arithmetic operations, nodes visited, etc.) which reflects in larger execution time compared to the speculative variants.

In general, the benchmarks SLS – J2D use a single eureka pattern and the `EX`, `CT`, and `EU` variants perform similarly. `EU` performs much better than the `FS` variant since the `EU` variant, like `EX` and `CT`, can trigger task cancellation even inside nested function calls. Overall, the

[2] `-XX:-UseGCOverheadLimit -Xmx65536m -XX:+UseParallelGC -XX:+UseParallelOldGC`.
[3] `if` checks and `return` happen only at the body of the `async`, not inside nested function calls.
[4] Our implementation minimizes overheads as it does not terminate worker threads, and it does not fill the stack trace of the abort exceptions.

Table 2 Benchmark execution time metrics, DNC means Did Not Complete inside 30 minutes. Except SUD, all the benchmarks had a coefficient of variation (CoV, ratio of the standard deviation to the arithmetic mean) less than 2 percent for the execution time of each variant. For SUD the CoV was about 10 percent for each variant.

Name	Execution Time (in seconds)					Ratio of Exec. Time				Abstract Operations ($\times 10^3$)				
	AF	FS	EX	CT	EU	AF:EU	FS:EU	EX:EU	CT:EU	AF	FS	EX	CT	EU
SLS	58.37	17.70	16.61	16.71	16.85	3.46	1.05	0.99	0.99	2,476	845	798	800	806
UTS	15.89	8.81	5.94	5.81	5.76	2.76	1.53	1.03	1.01	1,571	512	444	446	437
SUD	DNC	5.52	5.53	5.48	5.72		0.96	0.97	0.96		146	148	142	152
NQK	24.90	3.33	3.86	3.20	3.96	6.28	0.84	0.97	0.81	1,711	216	210	212	216
UTP	2.95	2.73	2.58	2.37	2.48	1.19	1.10	1.04	0.96	233	233	189	189	189
FLP	38.35	30.25	7.83	7.94	8.04	4.83	3.79	0.98	0.98	688	523	232	233	231
TSP	DNC	1.51	1.18	1.19	1.11		1.35	1.06	1.07		857	839	839	754
KMC	15.22	12.26	12.32	12.56	12.44	1.22	0.99	0.99	1.01	1,125	916	916	916	917
J2D	16.35	13.01	13.21	13.04	13.10	1.25	0.99	1.01	1.00	1,125	902	902	903	903
DLS-AND	169.67	50.94	47.67	48.15	47.87	3.54	1.06	1.00	1.01	500	172	164	162	163
DLS-OR	169.00	4.65	0.53	0.53	0.54	315.17	8.66	0.99	0.99	490	15	2	2	2
CS	7.33	7.36	7.44	7.55	2.86	2.56	2.57	2.60	2.64	360	360	360	360	135
						>4.15	1.53	1.08	1.06	>1,027	474	434	433	409
						Geometric Mean				Arithmetic Mean				

Table 3 Productivity metrics for benchmark kernels. LoC was computed using `cloc` command while CC and DE were computed using the CodePro Analytix Eclipse plugin. LoC for common support code are not reported in the table, the arithmetic mean for support code LoC is 240.

Name	Lines of Code					Cyclomatic Complexity					Development Effort ($\times 10^3$)				
	AF	FS	EX	CT	EU	AF	FS	EX	CT	EU	AF	FS	EX	CT	EU
SLS	72	75	79	78	69	1.66	1.77	1.88	1.88	1.55	11.22	12.16	14.84	12.97	10.8
UTS	76	84	88	87	81	1.50	1.70	1.80	1.80	1.60	7.63	11.66	12.74	12.15	9.2
SUD	86	94	98	97	92	1.60	1.80	1.90	1.90	1.70	15.62	21.55	23.49	22.61	19.0
NQK	81	87	94	93	85	1.60	1.70	1.80	1.80	1.60	16.12	18.90	22.91	20.72	17.5
UTP	85	101	105	104	91	1.70	1.81	1.90	1.90	1.54	11.63	19.84	24.30	21.34	13.7
FLP	115	115	116	115	108	1.91	2.00	2.08	2.08	2.00	62.89	64.42	65.47	65.97	70.9
TSP	89	101	105	104	99	1.60	1.80	1.90	1.90	1.54	22.28	33.79	35.83	35.28	27.8
KMC	115	127	128	127	135	1.38	1.69	1.69	1.69	1.46	46.57	51.24	55.98	51.24	56.8
J2D	146	152	156	155	148	1.64	1.78	1.85	1.85	1.53	111.70	116.83	127.58	119.18	104.4
DLS	80	85	89	88	79	1.88	2.22	2.33	2.33	1.66	16.80	19.46	22.42	19.99	17.4
CS	108	131	139	138	107	1.61	1.70	1.82	1.82	1.61	39.37	73.43	87.35	84.67	43.7
A. Mean	96	105	109	108	99	1.64	1.82	1.90	1.90	1.62	32.89	40.30	44.81	42.37	35.6
%age of EU	-3.75	5.30	9.41	8.41		1.63	12.25	17.76	17.76		-7.49	13.33	26.02	19.17	

`EU` variants compete favorably with the other speculative variants (`EX` and `CT`). On most benchmarks, the `EU`, `EX` and `CT` variants perform within 5% of each other, both in terms of execution time and the number of abstract operations. This shows that our `EU` abstractions and different `Eureka` patterns do not add significant overhead in their implementations. Note that our implementation uses delimited continuations without modifying the VM; the performance of our implementation would be greatly improved by using native support for DeConts in the VM. Work by Stadler et al. [42] to provide such native support in a Java VM reported over two orders magnitude speedup on micro-benchmarks compared to a bytecode transformation approach. Additionally, we decided to exclude benchmarks that further highlight the limitations of the other approaches (e.g. inaccessible functions, user exception handlers) to allow a fairer comparison between all the approaches. These benchmarks would show our `EU` approach in "an even more" positive light.

The DLS benchmark uses binary composition of `EU` `Eureka`s and performs similarly to `EX` and `CT` confirming that no significant overhead is introduced by the composition. The CS benchmark is interesting as the hierarchical nature of the computation allows the `EU` variant to terminate other tasks searching on different leading indices. The `CT`, `EX`, and `FS` variants cannot implement such hierarchical information easily and end up doing redundant computation even after the goal element has been found. This causes them to perform as much work (in terms of abstract operations) as the non-speculative version and causing larger execution times than the `EU` version.

Productivity Metrics Comparison: The most commonly used software productivity metric is program size or lines of code (LoC) to compare programs that use the same language and coding standards. There are other quantitative evaluation techniques for productivity apart from measuring LoC. McCabe introduced the Cyclomatic Complexity (CC) metric [28] based on the control flow structure of programs. CC represents the complexity of the algorithm, and poorly designed solutions have high CC values. Halstead's metrics [19] are also well-known measure of software complexity. The Development Effort (DE) metric represents the effort required to convert an algorithm to an actual code in a specific programming language.

We report values for LoC, CC, and DE for all our benchmark kernels in Table 3. We compare the metrics for the `AF` variants with four variants (`FS`, `EX`, `CT` and `EU`) which implement different task cancellation strategies. Overall, the `EU` versions require less LoC, CC, and DE being at least 5%, 12%, and 13% better, respectively, than any of the other speculative variants. More importantly, the percentage improvements are even larger when compared to the closest performing speculative variants – `EX` and `CT`. In addition, as explained in Section 3, the `EU` versions do not suffer from any of the drawbacks compared to the other speculative methods. On average, the `EU` solutions for the kernels are only slightly larger than the `AF` variants requiring three extra LoC, some extra DE (7.5%), while the CC is actually smaller. This shows that, for the benchmark kernels, minimal effort was required to transform the `AF` versions into speculative versions using our proposed model. In particular, the comparatively low value of DE for the `EU` variant also reflects positively upon the usability of the `Eureka` API. In summary, the `EU` solutions are more productive to implement than the `FS`, `EX`, and `CT` variants in terms of all three productivity metrics.

9 Related Work

Kolesnichenko et al. provide a comprehensive classification and evaluation of task termination techniques [24]. C# natively supports interruptive cancellation by throwing exceptions, and since the release of TPL also cooperative techniques [25]. Python supports interruptive cancellation of non-started tasks via executors and terminative cancellation of already started ones [37]. Java supports interruptive cancellations natively [34]. Pthreads library supports both termination and interruption of threads [3].

Burton [4] and Osborne [32] have both worked with speculative computation before. Burton proposes a deterministic feature that has simple semantics, i.e. produces the same result as a sequential computation. Osborne uses numerical priorities to order computations [32], in his work task priorities propagate among dependent (sponsored) tasks. The eureka scope of tasks is determined when they are stated ahead of time in `OR` clauses or as branches of a conditional. Computation termination is via the cancellation token approach where a programmer manually checks termination in each function. Compared to our model, Burton and Osborne style speculative execution support only the parallel search eureka pattern.

Prabhu et al. [36] propose two language constructs to declaratively express value speculation opportunities. Their approach relies on speculating the value of a computation and executing possible future computations that consume this value in parallel with the producer of the value. Our approach does not rely on value speculation and does not need to deal with the rollback of side-effects from *mispredicted* consumer tasks. Instead, we use speculative tasks in the EuPM to support a multitude of EuSCs.

Leaving the system in an inconsistent state is one of the problems with preemptive termination approaches. MVM [9] and J-SEAL2 [2] solve this problem by introducing isolation containers to segregate the data operated upon by tasks. Tasks cannot directly

share objects, and the only way for tasks to communicate is to use standard, copying communication mechanisms. Containers communicate via synchronous `receive` operations to pass notifications. Termination is effected on isolation containers by other containers; termination kills all worker threads assigned to individual containers. Our approach minimizes overheads as it avoids copying data, killing threads, and communicating via synchronous operations. In addition, creating containers is an expensive operation whereas, in our approach, creating multiple eureka sub-computations is cheap as it is akin to creating a task.

Cilk allows speculative work to be terminated through the use of Cilk's `abort` statement [15] inside function-scoped inlets. Cilk does not provide guarantees of when child tasks will be terminated, in fact, child tasks can be spawned even after the execution of an `abort` statement. However, the main difference is that in the EuPM only a subset of tasks can be terminated in contrast to terminating all child tasks via Cilk inlets. Perez and Malecha show several methods for implementing `abort` as a library in the Cilk++ system [35] by mechanically translating programs into continuation-passing style. Like our approach, spawned computations periodically poll to determine if they should terminate. While this transformation is simple, the problem with it is that it is not modular because it changes the signatures of functions that use the `abort` mechanism. This breaks the possibility for separate compilation without explicit annotations specifying which functions should be compiled to work with inlets and `abort`.

Ada offers a statement, `abort`, which allows a task to make abnormal any other visible task [16]. The `abort` statement will stop execution of the named task by the time it reaches a synchronization point, e.g. `delay` statement that suspends the execution of a task for some units of time. Unfortunately, the use of `delay` statements (even those with delays of 0.0) can be expensive operations, as each delay statement forces the runtime system to perform a context switch. In our approach, a task cannot directly cancel another task, it influences cancellation by triggering eureka events. Also, our `check` construct does not force a task to context switch, making it much cheaper to implement.

Both MPI and OpenMP support task grouping and cancellation. MPI provides termination support via the `MPI_Abort` function that terminates an MPI execution environment [45]. This function call makes a best attempt effort to terminate all tasks in the group of the communicator. OpenMP 4.0 API [31], released in July 2013, supports features to terminate parallel OpenMP execution cleanly. Tasks can be grouped to support deep task synchronization, and task groups can be terminated to reflect completion of cooperative tasking activities such as search. Threads check at user-defined cancellation points if cancellation has been requested. The cancellation points must be lexically nested in the type of construct specified in the clause; i.e. we cannot `cancel` from inside a method call. Our approach poses no such limitation on where a task can request cancellation and where the user-defined cancellation points can be placed in the program.

Tahan et al. [43] also propose a cancellation policy for OpenMP similar to Ada's abort where a task can cause the cancellation of a group of tasks (possibly not belonging to the same group as the currently executing task). In our approach a task can request cancellation of other tasks belonging to the same group. Like our approach, however, their approach also causes child tasks to inherit the cancellation properties from the parent task. Unlike our approach, certain tasks can be *protected* from being cancelled even though they belong to a cancelled group, and the task cancellation scheme is based on interrupts and exceptions. We chose to avoid such protected tasks to avoid any confusion and to keep the EuPM clean and simple.

10 Summary and Future Work

We introduced the Eureka parallel programming model (EuPM) that simplifies the expression of speculative parallel tasks in search and optimization algorithms. We have demonstrated the power of the EuPM as a mechanism for codifying various parallel eureka patterns. The constructs we propose simplify writing EuSCs and improve programmer productivity. Our implementation shows that the EuPM can also be implemented efficiently, especially with the need to terminate tasks cooperatively. Our performance results on benchmarks show that our implementation performs close to manual hand-coded versions while being shorter and less complex to write. We believe that our implementation techniques of the EuPM can easily be ported to other languages as well. We are planning to extend support for further eureka patterns and providing dynamic task priorities in EuSCs.

Availability

A supplementary artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). The artifact contains the source code of our library implementations of the different `Eureka`s and the different benchmarks used to simplify repeatability of all the experimental data. Public distributions of the `Eureka` implementation in Habanero-Java library are available for download at `http://wiki.rice.edu/confluence/display/PARPROG/HJ+Library`. The EuPM has also been taught in the introductory parallel programming class for second-year undergraduate students at Rice University (COMP 322). Additional documentation and code examples, are available in the course lecture and lab materials at `http://wiki.rice.edu/confluence/display/PARPROG/COMP322`.

Acknowledgments. We are grateful to the anonymous reviewers for their suggestions on improving the presentation of the paper. We would also like to thank Suguman Bansal, Brad Chamberlain, Prasanth Chatarasi, Tom Hildebrandt, Siam Hussain, Deepak Majeti, Sri Raj Paul, Alina Sbîrlea, Dragos Sbîrlea, and Hamim Zafar for their feedback on early drafts of this paper.

References

1 S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of molecular biology*, 215(3):403–410, October 1990.

2 Walter Binder. Design and Implementation of the J-SEAL2 Mobile Agent Kernel. In *SAINT'01*, pages 35–, 2001.

3 Bradford Nichols and Dick Buttlar and Jacqueline Proulx Farrell. *Pthreads Programming: Chapter 4 – Managing Pthreads*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

4 F. Warren Burton. Speculative computation, parallelism, and functional programming. *IEEE Trans. Computers*, 34(12):1190–1193, 1985.

5 Giorgio Buttazzo, Giuseppe Lipari, Luca Abeni, and Marco Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency*. Plenum Publishing Co., 2005.

6 Philippe Charles and et al. X10: An Object-Oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, 40:519–538, October 2005.

7 Liming Chen and A. Avizienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *FTCS-25*, Jun 1995.

8 Jens Clausen. Branch and Bound Algorithms – Principles and Examples. *Parallel Computing in Optimization*, pages 239–267, 1997.

9 Grzegorz Czajkowski and Laurent Daynés. Multitasking Without Comprimise: A Virtual Machine Evolution. In *OOPSLA'01*, pages 125–138, 2001.
10 Olivier Danvy and Andrzej Filinski. Abstracting Control. In *LFP'90*, pages 151–160, 1990.
11 Iulian Drago and et al. Continuations in the Java Virtual Machine. In *ICOOOLPS'2007*, 2007.
12 Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona OpenMP Tasks Suite. In *ICPP'09*, 2009.
13 Marc Feeley. Polling Efficiently on Stock Hardware. In *FPCA'93*, pages 179–187. ACM, 1993.
14 Mattias Felleisen. The Theory and Practice of First-Class Prompts. In *POPL'88*, 1988.
15 Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI'98*, pages 212–223, 1998.
16 J. Goldenberg and G. Levine. Ada's Abort Statement: License to Kill. *Ada Letters*, IX(6):97–103, September 1989.
17 Anne Greenbaum. *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.
18 Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *IPDPS'09*, pages 1–12, 2009.
19 Maurice H. Halstead. *Elements of Software Science*. Elsevier Science Inc., New York, NY, USA, 1977.
20 Christopher T. Haynes and Daniel P. Friedman. Engines Build Process Abstractions. In *LFP'84*, 1984.
21 Ilya Grigorik. Untangling Evented Code with Ruby Fibers. https://www.igvita.com/2010/03/22/untangling-evented-code-with-ruby-fibers/, 2010.
22 Shams Imam and Vivek Sarkar. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In *ECOOP'14*, 2014.
23 Shams Imam and Vivek Sarkar. Habanero-Java Library: a Java 8 Framework for Multicore Programming. In *PPPJ'14*. ACM, 2014.
24 Alexey Kolesnichenko, Sebastian Nanz, and Bertrand Meyer. How to Cancel a Task. In *Proceedings of MUSEPAT'13*, pages 61–72. Springer, 2013.
25 Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The Design of a Task Parallel Library. In *OOPSLA'09*, pages 227–242, 2009.
26 Wei-Ming Lin, Wei Xie, and Bo Yang. Performance Analysis for Parallel Solutions to Generic Search Problems. In *SAC'97*, pages 422–430, 1997.
27 Andrey Marochko. Exception Handling and Cancellation in TBB – Part II, May 2008.
28 T. J. McCabe. A Complexity Measure. *IEEE Trans. on Soft. Engineering*, 2(4), July 1976.
29 Oliver Kowalke. Introduction (Boost Coroutines). http://www.boost.org/doc/libs/1_53_0/libs/coroutine/doc/html/coroutine/intro.html, 2009.
30 Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. UTS: An Unbalanced Tree Search Benchmark. In *LCPC'06*, pages 235–250, 2007.
31 OpenMP API, Version 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, July 2013.
32 Randy B. Osborne. Speculative computation in multilisp. In *LFP'90*, pages 198–208, 1990.
33 W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *National Academy of Sciences of the United States of America*, 85(8):2444–2448, April 1988.
34 Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.
35 Ruben Perez and Gregory Malecha. Speculative Parallelism in Cilk++, 2012.

36 Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. Safe Programmable Speculative Parallelism. In *PLDI'10*, pages 50–61, 2010.

37 Python Software Foundation. concurrent.futures — Launching parallel tasks, August 2014.

38 Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient Data Race Detection for Async-Finish Parallelism. In *RV'10*, pages 368–383, 2010.

39 Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing First-class Polymorphic Delimited Continuations by a Type-directed Selective CPS-transform. In *ICFP'09*, 2009.

40 Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-Point Synchronization. In *ICS'08*, pages 277–288, 2008.

41 Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In *ECOOP'08*, 2008.

42 Lukas Stadler, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose. Lazy Continuations for Java Virtual Machines. In *PPPJ'09*, 2009.

43 Oussama Tahan, Mats Brorsson, and Mohamed Shawky. Introducing Task Cancellation to OpenMP. In *8th Int'l Workshop on OpenMP*, IWOMP 2012, pages 73–87, June 2012.

44 The Jikes RVM Project. Threading and Yieldpoints. `http://jikesrvm.org/Threading+and+Yieldpoints`, 2007.

45 The Open MPI Project. MPI_Abort. `https://www.open-mpi.org/doc/v1.8/man3/MPI_Abort.3.php`, 2014.

46 Richard Wiener. Branch and Bound Implementations for the Traveling Salesperson Problem. *Journal of Object Technology*, 2(2), 2003.

47 Wolfram. Solve Optimization Problems with Speculative Parallelism, November 2008.

Cooking the Books: Formalizing JMM Implementation Recipes*

Gustavo Petri[1], Jan Vitek[2], and Suresh Jagannathan[1]

1 **Purdue University, USA**
2 **Northeastern University, USA**

Abstract

The Java Memory Model (JMM) is intended to characterize the meaning of concurrent Java programs. Because of the model's complexity, however, its definition cannot be easily transplanted within an optimizing Java compiler, even though an important rationale for its design was to ensure Java compiler optimizations are not unduly hampered because of the language's concurrency features. In response, the *JSR-133 Cookbook for Compiler Writers* [15], an informal guide to realizing the principles underlying the JMM on different (relaxed-memory) platforms was developed. The goal of the cookbook is to give compiler writers a relatively simple, yet reasonably efficient, set of reordering-based recipes that satisfy JMM constraints.

In this paper, we present the first formalization of the cookbook, providing a semantic basis upon which the relationship between the recipes defined by the cookbook and the guarantees enforced by the JMM can be rigorously established. Notably, one artifact of our investigation is that the rules defined by the cookbook for compiling Java onto Power are *inconsistent* with the requirements of the JMM, a surprising result, and one which justifies our belief in the need for formally provable definitions to reason about sophisticated (and racy) concurrency patterns in Java, and their implementation on modern-day relaxed-memory hardware.

Our formalization enables simulation arguments between an architecture-independent intermediate representation of the kind suggested by [15] with machine abstractions for Power and x86. Moreover, we provide fixes for cookbook recipes that are inconsistent with the behaviors admitted by the target platform, and prove the correctness of these repairs.

1998 ACM Subject Classification D.1.3 Concurrent Programming, D.3.1 Formal Definitions and Theory, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Concurrency, Java, Memory Model, Relaxed-Memory

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.445

1 Introduction

A decade ago, the semantics of concurrent Java programs, the Java Memory Model (JMM), was revised and redefined [17]. This revision, which was adopted as part of the official Java specification [13] had multiple purposes. First, it was intended to replace the previous specification which disallowed many common architectural and compiler optimizations of Java programs that were found in many state-of-the-art JVMs. Second, it formalized, using a rather complicated axiomatic semantics, the possible behaviors of concurrent Java programs. Having a formalization, the *DRF-guarantee* [1] – establishing that programs that do not have data races (DRF) in their Sequentially Consistent (SC) semantics, can only exhibit SC behavior, even when executed on non-SC hardware – could be formally proved [5].

* This work is supported by the National Science Foundation under grants CCF-1216613 and CCF-1318227.

© Gustavo Petri, Jan Vitek, and Suresh Jagannathan;
licensed under Creative Commons License CC-BY

Unfortunately, due to the complexity of the formalism, many desirable properties of the semantics were not met, and many undesirable properties were not prevented [19]. In light of these shortcomings, there is currently a community effort to better understand and reconsider the definition of the JMM [12].

A testament to the complexity of the JMM specification is the *The JSR-133 Cookbook for Compiler Writers* [15], an informal guide to implementing the JMM in different computer architectures. This document is intended to aid Java compiler writers to provide safe, reasonably efficient implementations, that nonetheless satisfy the JMM requirements. Unlike the JMM, the high-level semantics of Java concurrency is described operationally, in terms of memory instruction reorderings, thus defining the relaxed behaviors a program may exhibit, in a form suitable for reasoning about the correctness of compiler optimizations.

One of the reasons why the current JMM specification is so complex is that it attempts to uniformly capture the set of memory relaxations induced by both relaxed-memory platforms as well as common compiler optimizations deemed necessary to provide performant Java implementations. A recent effort [9] has considered an alternative approach, namely giving a semantics to Java that captures only the relaxations permitted by the TSO (Total Store Ordering) memory model found on x86 architectures [21]. One could attempt to implement this flavor of Java in weaker architectures such as Power [24], but this is a substantially more challenging exercise; simply retrofitting the TSO-aware semantics developed in [9] for Power would incur a high performance cost, necessitating injection of low-level synchronization operations between normal variable memory accesses to ensure TSO behavior.

The following question thus presents itself: what is the strongest memory model that would be both (1) efficiently implementable – not requiring synchronization at the low level for non-volatile variables – in architectures as relaxed as Power, and (2) yet have a tractable formal semantics amenable to the rigorous proofs needed to demonstrate compiler correctness arguments à la CompcertTSO [20]? As a corollary, we also wish to understand the semantics of current *implementations* of JVMs with respect to the memory model they support. JVMs ensure their implementations are consistent with the JMM by making conservative decisions on synchronization and shared-memory accesses. We are interested in determining if there is a middle ground between the behaviors admitted by relaxed-memory architectures and the JMM, which provides a more tractable, perhaps stronger semantics than the JMM, but which provides nonetheless an acceptable performance for modern Java applications.

At first glance, it would appear that many of these questions are answered in [15]. However, given that [15] is an informal document, with no clear – let alone formal – semantic definitions, and no guarantees that the rules defined are correct, we consider a methodology to formalize the semantics induced by its "recipes", deriving as an important by-product, a provable validation that some of the minimal guarantees required by the JMM are satisfied. In this sense, our goals are broadly similar to [6], which provides a provably correct compilation strategy of C++11 into Power. However, operating as we do in the Java context, our challenges are substantially different; not only must our formalization cope uniformly with different architectures given the platform agnostic definition of the JMM, but it must also deal explicitly with a number of JMM-specific features such as its support for "roach-motel" reorderings, explicitly established as a requirement of the JMM [17]. These issues make it infeasible to seamlessly transplant the results from approaches like [6]. Unlike [6], we do not provide a concrete compilation strategy – indicating for example that a fence has to be emitted *immediately after* a volatile store – but rather indicate minimal constraints that must be satisfied by any such strategy – for example a fence must exists in between a volatile store and any subsequent memory action –. We do this to allow flexibility to capture systems

like Octet [7] where the fences might be added in garbage collection safe points for example. This follows the spirit of [15].

Perhaps surprisingly, the relation between [17] and [15] has not been considered formally before, and notably our results show that the rules implied by [15] for Power are at odds with the requirements of the JMM.[1] Concretely, while working on our proofs we found a counter-example to the DRF requirement of the JMM if the rules of [15] are used for Power. The example in question is the infamous IRIW litmus test – reproduced below – considering only *volatile* variables instead of normal variables. In Java, concurrent conflicting accesses to volatile variables are not considered to form a data races. We display the example below with each thread in a column, and we consider that the object o is shared among all threads, with volatile fields v and w. Variables starting with r are local to each thread.

$o.v = o.w = 0$ & both fields are volatile			
$o.v = 1;$ ‖	$o.w = 1;$ ‖	$r0 = o.v;$ ‖	$r2 = o.w;$
‖	‖	$r1 = o.w;$ ‖	$r3 = o.v;$
Is $r0 = r2 = 1$ & $r1 = r3 = 0$ allowed?			

The behavior in question cannot be produced by an SC semantics. However, this behavior is possible in Power [24]. Moreover, inserting lwsync Power barriers in between the two reads in the reading threads would not prevent this behavior from happening as documented in [24, 8].[2] Unfortunately, lwsync was the barrier of choice recommended by [15] when our work was started to prevent this relaxation.[3] We tried this Java example in a Power 7 machine, and were able to reproduce the erroneous behavior in the two different JVM's we tested[4], indicating that this is not simply a theoretical inconvenience, but a critical dichotomy between desired semantics and implementations. Our discussions with several VM implementors indicate that (a) the cookbook was heavily used as a crucial reference, given the complexity of the official specification, and (b) some implementations are aware of the bug noted above, while others are not; given the subtlety and complexity of the JMM, and the lack of consensus among implementors on a proper implementation strategy, the anecdotal evidence makes clear that a cookbook-like document is quite necessary, with a provably correct version even more so. To highlight the subtlety of the issues involved, parts of the cookbook were in fact changed [6] in response to advances in the formalization of processor memory models (e.g., [16, 24]), but in the absence of a formal definition, those changes did not remediate the issues noted here.

The contributions of this paper are:

1. We formalize (operationally) the semantics of compiling concurrency features in Java as described by [15] into the x86 and Power relaxed-memory architectures.
2. Notably, our high-level semantics propagates the relaxations admitted by Power to normal Java variables. Our choice to propagate Power semantics for normal variables into a high-level semantics is motivated by the fact that any stronger semantics at the high-level would impose synchronization operations for normal variables in Power. This would most likely greatly degrade the performance of concurrent Java programs in Power, which is on

[1] Of course, many of the architectures considered in [15] were not as well understood by the research community at the time it was published.
[2] The behavior manifests because lwsync imposes no constraints on when the stores performed by the first two threads become visible to the readers.
[3] At the time of this writing, December 2014, the cookbook has been updated based on our findings.
[4] The example failed on IBM's JVM and Jikes RVM. Similar examples failed in Fiji's realtime JVM implementation on ARM 7.

the one hand unnecessary given the JMM definition, and on the other hand not required by [15]. We consider this to be a minimal performance requirement for any acceptably efficient implementation of the JMM on Power. Given that Power is one of the weakest architectural memory models yet studied, we consider that our high-level semantics serves as an upper bound of how strong a JMM could be, without penalizing weak architectures like Power.

3. [15] uses an intermediate representation to express memory operation reorderings. We formalize this intermediate representation, and prove a simulation argument between source-level programs and programs compiled to this IR, and establish an inclusion property between behaviors allowed by the target architectures (x86 and Power) and this IR.
4. We additionally formalize the different target architectures we consider in the same framework, and when the rules of [15] are correct we prove that they are so. Additionally, we identify the rules that *do not produce correct implementations*, and propose corrections, which we then prove sufficient to enforce the expected high-level semantics (e.g., volatile variables must exhibit SC semantics). Our findings have been propagated to the current revision of [15].
5. To the best of our knowledge, ours is the first formal attempt to relate the high-level semantics of the JMM with low-level architectural implementations as described in [15].

We emphasize that *it is not our aim to provide a new memory model for Java* – which presumably would be weaker than our IR to allow for additional relaxations –, instead we are simply using [15] as a harness for how existing implementations manifest the rules of the JMM. In short, we are documenting the ad-hoc model that implementors use.

The remainder of the paper is organized as follows. The next section provides additional motivation and gives an overview of our approach and proof structure. Section 3 presents the syntax and single-threaded semantics of the core language studied in terms of an abstract machine that admits weak memory behavior. Section 4 extends the semantics to deal with concurrency features found in *cookbook-high*, a language that supports normal references (with a Power-style relaxation) and volatile references, as well as locks. Section 5 describes a low-level intermediate representation (*cookbook-low*) that implements memory barriers, and does not support volatile references. We define the conditions necessary to compile cookbook-high into this IR, and provide a simulation result between executions in the low and high languages. Section 6 defines the semantics for x86 and Power in terms of our core language, and establishes a correspondence between cookbook-low programs and programs compiled to these architectures. Related work and conclusions are given in Sections 7 and 8, resp.

2 Overview

Consider the requirements of the JMM with respect to the implementation of synchronization operations, and its relation to the rules provided by the cookbook document. A driving principle of the JMM, dubbed the *roach motel semantics* [17], is that increasing the synchronization of a program cannot *add* new observable behaviors to it. The synchronization operations, formally defined in [17], include locking and volatile memory access operations.[5]

[5] Thread creation, termination, and object initialization are also synchronization operations, but they are not relevant for the ideas discussed here.

Table 1 High-level Roach-Motel Semantics Rules.

1st Op.\2nd Op.	Normal Load / Store	Volatile Load / Lock	Volatile Store / Unlock
Normal Load / Store			No
Volatile Load / Lock	No	No	No
Volatile Store / Unlock		No	No

The roach motel principle implies that all program transformations which increase the *happens-before* [14] relation of the program – which captures the causality relation of a program enforced through its synchronization actions (locks and volatile accesses) – should be allowed by the memory model. Pragmatically, this means that normal memory operations following a volatile write can be reordered before it, since the resulting program imposes additional synchronization not required by the former. Similarly, normal memory operations preceding a volatile read can be reordered after it. An argument similar to the case of volatile writes applies to unlock operations (a monitorexit in Java bytecote), and the same is true for volatile reads with respect to lock operations (monitorenter). These observations justify the first table presented in the cookbook [15], that describes the reorderings possible at the highest-level considered in that document. We reproduce this table in Table 1. The table indicates that two operations can be reordered if the cell is empty, and that they cannot if the cell is marked "No"; the first operation is sampled from the rows and the second one from the columns. Data and control dependencies are assumed to be respected by the cookbook tables. Then, for instance two normal memory operations on different references can be freely reordered, but any two synchronization operations cannot.

Intermediate Representation

Before presenting the requirements for the implementation of these operations for a specific architecture, the cookbook introduces an intermediate low-level representation in which memory operations are not assumed to have inherent ordering semantics; instead, operation ordering is imposed through the use of additional barrier – or fence – instructions, that guard the kind of reordering permissible between two memory accesses. At this level, volatile memory operations are assumed to be "implemented" using normal memory operations – corresponding to the operations provided by the ISA of the target architectures –, and the ordering constraints of Table 1 have to be enforced rather than assumed. This intermediate representation assumes that there is a different barrier to prevent the reordering of any two kind of memory operations if the barrier is emitted by the code in between these two accesses. For example, two read operations can be prevented from being reordered if a *Load to Load* barrier (LoadLoad) is emitted in between them by the thread. Similar fences exist between stores and loads, loads and stores and two consecutive stores. Table 2 presents the kind of barriers that must be introduced in this intermediate representation to enforce the semantics of Java delineated by Table 1. This is the second table of [15].

Given the lack of a precise semantics for normal load and store instructions, it is difficult to formally establish the correspondence between the high- and low-level versions. Our first contribution (section 4) is the definition of a tractable semantics for these two layers that enables the correctness proof of the rules relating these two tables.

In [15], tables are presented which for each architecture relate the instructions from the corresponding ISA to implement each of the barriers described above. We postpone the discussion of how we establish the correspondence between low-level cookbook barriers

Table 2 Low-level Cookbook: Barriers Required.

1st Op.\2nd Op.	Normal Load	Normal Store	Volatile Load/Lock	Volatile Store/Unlock
Normal Load				LoadStore
Normal Store				StoreStore
Volatile Load/Lock	LoadLoad	LoadStore	LoadLoad	LoadStore
Volatile Store/Unlock			StoreLoad	StoreStore

and architecture-specific instructions until section 6. Notably this final table provides only translations for the barrier instructions, leaving normal operations unrestricted.

Store-Atomicity Relaxation

A limitation of the cookbook document is that the argumentation is made in terms of operation reorderings, which disregards *store-atomicity* – or write-atomicity – which allows write operations to be propagated to different threads at different times, a relaxation permitted by some architectures, including Power and ARM [24, 3]. One could imagine providing a semantics which considers reordering of operations as the only source of relaxations in the style of the TSO, PSO and RMO [27] memory models. However, this would be insufficient to capture certain important relaxations that are permitted by architectures with weaker memory models; the following example (similar to the example WRC of [24]) illustrates this issue.

$$\begin{array}{c} o.f = o'.f = \mathtt{NULL} \\ \hline \begin{array}{c|c|c} o.f = o' & (o.f).f = o & r0 = o'.f; \\ & & r1 = r0.f \end{array} \\ \hline r0 = o \ \& \ r1 = \mathtt{NULL}? \end{array} \tag{1}$$

This program has three threads, which share two objects o and o', each with a single field f initially NULL. We assume that the type of the field f is the same as the type of o and o'. In the result indicated at the end, we have that $r0 = o$, therefore it must be the case that the read of $o'.f$ in the third thread returns the object o. Indeed this is possible if the first thread executes first, then the second thread dereferences $o.f$ obtaining o' and after that it writes o into $o'.f$. Now we can fullfil the read of $r0$ in the third thread. It is obvious that the read of $r0.f$ in the third thread cannot happen before $r0$ has obtained its value through the previous read. Therefore these two reads cannot be reordered. In that case, if the only source of relaxation is reordering, the read $r0.f$ which in actuality is a read of $o.f$ must see the value o', since all reorderings are prevented through data dependencies. This final result cannot be produced by a *reordering-only* memory model. However, this is a possible behavior in Power, since a write-atomicity relaxation could mean that the write of the first thread is only propagated to the second, but not the third thread, allowing the third thread to read NULL for $r1$. To admit such behavior, it is then necessary to introduce write-atomicity relaxations existent in Power within the (low-level) cookbook semantics to avoid over-synchronizing normal memory accesses. This motivates the semantics we present in section 4.

Proof Structure

Figure 1 illustrates the overall proof structure that we follow in our work. At the top level, we have the semantics of the JMM as described in [17], or rather the improved version of [19].

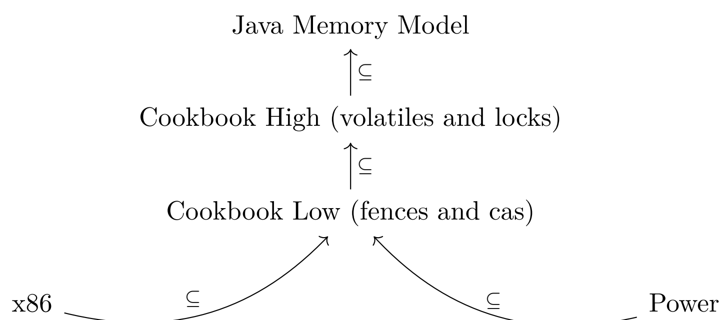


Figure 1 Models above PowerMM exhibit Write-Atomicity Relaxations.

Below this level, we have a high-level, architecture-agnostic, operational semantics which adopts Power semantics for normal variables, and SC semantics for volatile variables and locks. We denote this semantics by cookbook-high. One level down, we have the intermediate representation that contains only normal memory accesses and barriers. Finally, at the bottom of the figure we have the semantics of the Power and x86 architectures, of which Power offers a more relaxed semantics. We establish a backwards simulation between the high and low-level definitions of the cookbook, show that high-level cookbook semantics respects the JMM, and that our low-level cookbook definition properly captures the behaviors admitted by x86 and Power.

In the next section we will introduce our unifying language and semantic artifacts that make our simulations and proofs possible.

3 A Language

We define an operational semantics for a relaxed memory framework inspired by [8]. We describe different memory models using the same basic syntax for the different languages discussed in the previous section. For example, the semantics of the top-level language that we consider (akin to a Java bytecode) includes a treatment of volatile variables; volatiles, however, do not appear in lower-level languages. Additionally, the languages that model specific architectures add barrier instructions, which are not present in higher-level languages.

We first introduce the main languages considered:

COOKBOOK-HIGH: This is the top-level language, and is intended to model the memory-model related concerns of a Java-bytecode like language. As such it contains: (a) *Normal references*, which are the normal fields and variables of a Java program; (b) *Volatile references*, which are variables subject to strict ordering and visibility constraints as dictated by [17]. For example, volatile variables should have SC semantics when considered in isolation; and, (c) *Locks* used to represent the mutually-exclusive lock in a Java monitor. As with volatile references, locks are subject to strict visibility and ordering constraints [17]. We do not concern ourselves with a proper definition of "object" in this work, since this notion is irrelevant for the memory-model issues being studied. We reiterate that this language, keeping in the spirit of Java bytecode, contains no barrier (or fence) operations.

COOKBOOK-LOW: This is an intermediate representation used in [15] to establish the barriers needed to impose ordering at lower-levels to *implement* the semantics of cookbook-high. This language serves to bridge the gap between the cookbook-high language and multiple target architectures, each of which have their own ISA that implements different barriers

and memory models. The main difference between cookbook-low w.r.t. cookbook-high are: (a) cookbook-low has no notion of volatile reference. References that are volatile at the top-level are considered as ordinary normal references in this level, and (b) cookbook-low provides barrier instructions, to prevent the local reordering of certain type of instructions. For instance, the syntax $\langle \texttt{l}|\texttt{s} \rangle$ in this language is used to guarantee that a load memory accesses (i.e. a read) issued by a thread prior to the execution of the barrier cannot be performed later than any store operation (i.e. write) issued after the barrier by the same thread. This language includes all the barriers presented in [15]. This is a common intermediate representation that is subsequently compiled to each different target architecture.

Power (PPC): This language, although expressed as a functional core, is intended to model the Power architecture as documented in [24, 8]. The main differences between this language and cookbook-low are: (a) this language implements the actual barrier instructions of Power, that is lwsync and sync as opposed to the more abstract barriers of cookbook-low[6], and (b) unlike the barriers of cookbook-low, these barriers have a global meaning (potentially involving more than one thread) as documented in [24, 8].

TSO (x86): This language represents the TSO memory model of x86 processors [21]. It has only one barrier instruction, namely mfence. It also has a cas() instruction, used in the implementation of locks.

3.1 Syntax

The syntax of our core language is a simple first-order language equipped with references, volatile references for cookbook-high, locks, conditionals and boolean values and operators. As mentioned earlier, we have different barrier instructions at different levels of our languages, which are part of the syntax. The syntax of our language is in ANF [11] to simplify the definition of evaluation contexts, and sequencing of operations, which is irrelevant for our purposes. Our source level syntax involves the following semantic categories:

$$x, y \in \mathcal{V}ar \qquad p \in \mathcal{P}tr \qquad \mathsf{p} \in \mathcal{V}ptr \qquad \ell \in Locks$$

$\mathcal{V}ar$ represent variables with substitution semantics, $\mathcal{P}tr$ represent normal pointers, $\mathcal{V}ptr$ represent *volatile* pointers, and $\mathcal{L}ocks$ represent locks.

We present the syntax of our language in Figure 2. As customary, the values of the language, represented by the set $\mathcal{V}al$, include variables (a convenience to have our language in ANF), booleans, references, volatile references, locks and a special unit value () to represent termination. Expressions in our language, represented by the set $\mathcal{E}xpr$ contain all values, and boolean operators, which are implicit and ranged over by the metavariable $\oplus$. Commands include the standard `skip`, sequence and a simple let-binding construct to evaluate complex boolean expressions. Moreover, we have standard conditionals, reference creation, assignment, and dereferencing.

For the cookbook-high language, we include a number of commands (in red) which operate on volatile variables and locks. Unlike Java, cookbook-high has special syntax to operate over volatile references. We do not consider programs that use the volatile syntax to access normal references, nor the converse.[7] This assumption will be made precise when presenting

[6] We only consider a subset (namely the ones needed by [15]) of available Power barriers in our development.

[7] In Java the distinction between volatile and normal memory accesses can be made through the *type* of a field. Here we assume a distinguished syntactic form, and implicitly consider only programs that make consistent assumptions in the syntax and runtime about volatile accesses.

$$
\begin{array}{rcl}
v \in \mathcal{V}al & ::= & x \mid tt \mid f\!f \mid p \mid \mathsf{p} \mid \ell \mid () \\
e \in \mathcal{E}xpr & ::= & v \mid e \oplus e \\
\mathsf{c} \in \mathcal{L} & ::= & v \mid \texttt{skip} \mid \mathsf{c}_0 \,;\, \mathsf{c}_1 \\
& \mid & \texttt{let}\ x = e\ \texttt{in}\ \mathsf{c} \\
& \mid & \texttt{if}\ v\ \texttt{then}\ \mathsf{c}_0\ \texttt{else}\ \mathsf{c}_1\ \texttt{fi} \\
& \mid & \texttt{let}\ x = \texttt{new}\ v\ \texttt{in}\ \mathsf{c} \\
& \mid & v_0 := v_1 \\
& \mid & \texttt{let}\ x = !y\ \texttt{in}\ \mathsf{c} \\
& \mid & \texttt{let}\ x = \texttt{new}_v\ v\ \texttt{in}\ \mathsf{c} \\
& \mid & v_0 :=_v v_1 \\
& \mid & \texttt{let}\ x = !_v y\ \texttt{in}\ \mathsf{c} \\
& \mid & \texttt{lock}\ x \mid \texttt{unlock}\ x \mid b
\end{array}
$$

$$
\begin{array}{rcl}
b \in \mathcal{S}ync\text{CBL} & ::= & \langle \mathtt{s}|\mathtt{l}\rangle \mid \langle \mathtt{l}|\mathtt{l}\rangle \\
& \mid & \langle \mathtt{l}|\mathtt{s}\rangle \mid \langle \mathtt{s}|\mathtt{s}\rangle \\
b \in \mathcal{S}ync\text{TSO} & ::= & \mathsf{mfence} \\
& \mid & \texttt{let}\ x = \mathsf{cas}(y)\ \texttt{in}\ \mathsf{c} \\
b \in \mathcal{S}ync\text{PPC} & ::= & \mathsf{lwsync} \\
& \mid & \mathsf{sync}
\end{array}
$$

Figure 2 Syntax. The statements in red are present in Cookbook-High only. The synchronization statements b in blue depend on the language being considered.

the cookbook-high memory semantics. The first three commands in red create, store and read a volatile reference, respectively. The commands `lock` x and `unlock` x assume that the variable x will adopt a lock value at runtime and roughly represent the `monitorenter` and `monitorexit` bytecode instructions of Java.

Finally, the languages at levels lower than cookbook-high contain a number of barrier instructions. We add these to the syntax, and will restrict their usage according to the language being considered. In the case of `x86`, we consider a `cas`() instruction that atomically queries and updates a memory location. In our restricted language, the argument x to $\mathsf{cas}(x)$ is assumed to be a reference, which if it is initially $f\!f$ will be set to tt, returning tt; otherwise, the operation has no effect, and returns $f\!f$.

3.2 Semantics

Our semantics follows [8] which models states in terms of configurations with rewriting rules that dictate how programs can reduce or perform effects on that state. The state is a triple, $(\sigma, \delta, \mathsf{T})$ comprising: (1) a store σ which is a mapping from references (and volatile references) to their current value, (2) a thread system T, which is a mapping from thread identifiers – sampled from the set $\mathcal{T}id$ and ranged with the metavariable t – to runtime commands (i.e. elements of $\mathcal{L}$ where some variables have been substituted by runtime values). These commands represent the continuation of the original command of thread t, and finally (3) what we shall call a *temporary store*, which is represented by the metavariable δ. A temporary store is a sequence[8] of pending *memory operations* associated with their originating thread identifiers. They represent operations that have been issued by the threads, but not yet fully synchronized with the memory (σ), and the other threads. In essence, an operation in δ, is an operation that has been issued, perhaps has been partially executed, potentially made visible to some threads but not all, which the memory system will have to commit at a later point in time. Different rules for committing the operations in the temporary store allow us to define different *ordering* and *visibility* components that collectively define a memory model.

[8] We use the notation $\delta \cdot \delta'$ for sequence concatenation, and ϵ to denote the empty sequence.

MEMORY OPERATIONS

$$\mathsf{mo} \in \mathcal{O}pr \;::=\; \mathsf{t}^v_\rho \mid \mathsf{e}^v_\rho \mid \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{\varrho,\mu} \mid \mathsf{rd}_{\varrho,\mu} \mid \overline{\mathsf{rd}_{\rho,\mu}} \mid \mathbf{vwr}_{\varrho,\mu} \mid \mathbf{vrd}_{\varrho,\mu} \mid \overline{\mathbf{vrd}_{\rho,\mu}} \mid \mathtt{lk}_\mu \mid \mathtt{ul}_\mu \mid b$$

SINGLE-THREAD STEP

$$\frac{[\![r]\!](\sigma,\delta,t) = (\mathsf{mo},\mathsf{c})}{(\sigma,\delta,(t,\mathbf{E}[r])\|\mathsf{T}) \xrightarrow[t]{\mathsf{mo}} (\sigma,\delta\cdot(t,\mathsf{mo}),(t,\mathbf{E}[\mathsf{c}'])\|\mathsf{T})}$$

$$[\![r]\!](\sigma,\delta,t) = \begin{cases} (\tau,()) & \text{if } r = \mathtt{skip} \\ (\tau,\mathsf{c}) & \text{if } r = v\,;\mathsf{c} \\ (\mathsf{t}_\rho,\mathsf{c}_0) & \text{if } r = (\mathsf{if}\ \rho\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1) \\ (\mathsf{e}_\rho,\mathsf{c}_1) & \text{if } r = (\mathsf{if}\ \rho\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1) \\ (\mathsf{t}^{tt},\mathsf{c}_0) & \text{if } r = (\mathsf{if}\ tt\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1) \\ (\mathsf{e}^{ff},\mathsf{c}_1) & \text{if } r = (\mathsf{if}\ ff\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1) \\ (\tau,\mathsf{c}[x \leftarrow \mu]) & \text{if } r = \mathtt{let}\ x = \mu\ \mathtt{in}\ \mathsf{c} \\ (\mathsf{rd}_{\varrho,\rho},\mathsf{c}[x \leftarrow \rho]) & \text{if } r = \mathtt{let}\ x = !\varrho\ \mathtt{in}\ \mathsf{c}\ \&\ \text{free}\ \rho\ \text{in}\ \delta \\ (\mathbf{vrd}_{\varrho,\rho},\mathsf{c}[x \leftarrow \rho]) & \text{if } r = \mathtt{let}\ x = !_v\varrho\ \mathtt{in}\ \mathsf{c}\ \&\ \text{free}\ \rho\ \text{in}\ \delta \\ (\mathsf{wr}^{\{t\}}_{\varrho,\mu},()) & \text{if } r = (\varrho := \mu) \\ (\mathbf{vwr}_{\varrho,\mu},()) & \text{if } r = (\varrho :=_v \mu) \\ (\mathsf{wr}^{\{t\}}_{p,\mu},\mathsf{c}[x \leftarrow p]) & \text{if } r = \mathtt{let}\ x = \mathtt{new}\ \mu\ \mathtt{in}\ \mathsf{c}\ \&\ \text{free}\ p\ \text{in}\ \delta,\sigma \\ (\mathbf{vwr}_{\mathsf{p},v},\mathsf{c}[x \leftarrow \mathsf{p}]) & \text{if } r = \mathtt{let}\ x = \mathtt{new}_v\ \mu\ \mathtt{in}\ \mathsf{c}\ \&\ \text{free}\ \mathsf{p}\ \text{in}\ \delta,\sigma \\ (b,()) & \text{if } r = b \\ (\mathtt{lk}_\mu,()) & \text{if } r = \mathtt{lock}\ \mu \\ (\mathtt{ul}_\mu,()) & \text{if } r = \mathtt{unlock}\ \mu \end{cases}$$

Figure 3 Single thread semantics. Thread composition (no memory actions).

The dynamic semantics captured by this framework thus allows (a) threads to contribute (by executing their code) memory operations into the temporary store δ; and (b) the memory system to take care of *committing* these operations in the main memory σ, and synchronizing the memory operations of all threads.

Intra-thread Semantics

The contribution of each thread to the temporary store is presented as a reduction semantics decomposing each command into a *reducible expression* (redex) and an evaluation context. Our semantics preserves the invariant that at runtime each command can be decomposed into a unique evaluation context and a redex, or it contains an error, in which case we disregard the computation. Below is our definition of evaluation contexts, and evaluation context application.

$$\mathbf{E} ::= [] \mid \mathbf{E}\,;\mathsf{c} \qquad\qquad \mathbf{E}[\mathsf{c}] = \begin{cases} \mathsf{c} & \text{if } \mathbf{E} = [] \\ \mathbf{E}'[\mathsf{c}]\,;\mathsf{c}' & \text{if } \mathbf{E} = \mathbf{E}'\,;\mathsf{c}' \end{cases}$$

To the set of values presented in Figure 2 we add a category of runtime *placeholder* values used to delay the effects of reads without blocking the execution of subsequent instructions in the program (these are called *Identifiers* in [8]).

$$\rho \in \mathcal{P}l\mathcal{H}old \qquad \varrho \in \mathcal{P}l\mathcal{H}old\mathcal{P}tr \;::=\; p \mid \mathsf{p} \mid \rho \qquad \mu \in \mathcal{P}hold\mathcal{V}al \;::=\; v \mid \rho$$

The set $\mathcal{P}lHold$ contains an infinite set of values – ranged over by ρ – that will be used at runtime to stand in place of an actual value returned by a read (similar to the semantics of futures in [10]). We will use the metavariable ϱ to range over placeholder values or reference values (both normal and volatile), which can appear in the left hand side of an assignment, or in a dereferencing instruction. We will use the more general metavariable μ to range over placeholders or any other value, which can appear in the program anywhere a value is expected.

Using these placeholders, each time a read redex is to be reduced, we do not immediately query the memory, but instead generate a read operation in the temporary store, where a fresh placeholder takes the place of the value that will be queried at a later point in the execution. Hence, our semantics preserves the invariant that any placeholder value appearing in a program must have been generated by a prior read operation, which is still uncommitted (i.e. the memory system has not returned a value for it yet). Once the read operation is committed, all placeholders are replaced with the appropriate value. Finally, we remark that *placeholders are not storable values*. Thus, writes to memory can only be committed if they contain an actual value in $\mathcal{V}al$. The top left of Figure 3 defines the contributions of each thread to the temporary store.

Our language permits a light form of branch speculation, achieved by predicting a branch when a placeholder value is the condition of an `if` instruction. An operation $\mathtt{t}^{v}_{\rho}$ represent the speculation of a *then* branch (where any of v or ρ could be absent). This operation is contributed by a thread taking a *then* branch, where the condition of the branch is a placeholder (ρ). If the conditional is evaluated on a value (v) instead, $\mathtt{t}^{v}$ is produced, which is not really a speculation, since the value of the condition is known. In the case of a real speculation, at a later point, the value of placeholder ρ will be substituted (say by v), and the operation will be substituted by $\mathtt{t}^{v}_{\rho}$ (we need to keep track of the placeholder to enforce ordering constraints w.r.t. speculations). Clearly, if v is a *ff* this will be a mis-speculation since we are considering a then branch, and we will simply disregard the miss-predicted trace. Note that when the condition of a branch is a placeholder, a branch can always proceed given that the placeholder is later substituted with a value that matches the prediction (i.e. $\mathtt{t}^{tt}_{\rho}$). Similarly $\mathtt{e}^{v}_{\rho}$ represents the speculation of an *else* branch.

The second line of operations corresponds to memory operations on *normal* references. For the time being, we disregard the action $\overline{\mathsf{rd}_{\rho,\mu}}$ which will be considered when presenting the semantics of the memory system. The first action, $\mathsf{wr}^{\mathcal{W};\mathcal{I}}_{\varrho,\mu}$ is a write action emitted by a thread. The components ϱ and μ are the reference (or placeholder) that is being written, and the value (or placeholder) that is being written into it. Additionally, to capture the semantics of atomicity relaxations of Power, each write operation in the temporary store contains a set $\mathcal{W}$ of *thread identifiers*, indicating which thread can currently see this write – even before the write is executed in the memory. For instance, a write event $\mathsf{wr}^{\mathcal{W}\cup\{t\}}_{p,v}$ can be seen by reads of thread t before reaching the memory (σ). When threads emit write events the set $\mathcal{W}$ contains only the current thread $\{t\}$ (simulating store-buffers à la TSO). Similarly, the set $\mathcal{I}$ represents a set of placeholder values whose originating read has been fulfilled by this write. This component is only used to give semantics to Power barriers. Throughout the paper, whenever any of these sets are empty we will omit them for readability. Read memory operations $\mathsf{rd}_{\varrho,\mu}$ correspond to the issuance of a read operation, on reference (or placeholder) ϱ whose return value (or placeholder) is μ. Initially, μ will always be a fresh placeholder value. Later, the memory system will substitute this placeholder by a concrete value read from memory.

$$
\begin{array}{rcll}
(\sigma, (t, \mathtt{t}_{tt}) \cdot \delta, \mathsf{T}) & \underset{\text{SC}}{\hookrightarrow} & (\sigma, \delta, \mathsf{T}) & \\
(\sigma, (t, \mathsf{e}_{ff}) \cdot \delta, \mathsf{T}) & \underset{\text{SC}}{\hookrightarrow} & (\sigma, \delta, \mathsf{T}) & \\
(\sigma, \delta_0 \cdot (t, \mathsf{wr}_{p,v}) \cdot \delta_1, \mathsf{T}) & \underset{\text{SC}}{\hookrightarrow} & (\sigma[p \leftarrow v], \delta_0 \cdot \delta_1, \mathsf{T}) & \left[\delta_0 \overset{\frown}{\text{SC}} (t, \mathsf{wr}_{p,\rho})\right] \\
(\sigma, \delta_0 \cdot (t, \mathsf{rd}_{p,\rho}) \cdot \delta_1, \mathsf{T}) & \underset{\text{SC}}{\hookrightarrow} & (\sigma, \delta_0 \cdot (\delta_1[\rho \leftarrow v]), \mathsf{T}[\rho \leftarrow v]) & \left[\delta_0 \overset{\frown}{\text{SC}} (t, \mathsf{rd}_{p,\rho}) \ \&\ \sigma(p) = v\right] \\
(\sigma, \delta_0 \cdot (t, \mathbf{vwr}_{p,v}) \cdot \delta_1, \mathsf{T}) & \underset{\text{SC}}{\hookrightarrow} & (\sigma[p \leftarrow v], \delta_0 \cdot \delta_1, \mathsf{T}) & \left[\delta_0 \overset{\frown}{\text{SC}} (t, \mathbf{vwr}_{p,\rho})\right] \\
(\sigma, \delta_0 \cdot (t, \mathbf{vrd}_{p,\rho}) \cdot \delta_1, \mathsf{T}) & \underset{\text{SC}}{\hookrightarrow} & (\sigma, \delta_0 \cdot (\delta_1[\rho \leftarrow v]), \mathsf{T}[\rho \leftarrow v]) & \left[\delta_0 \overset{\frown}{\text{SC}} (t, \mathbf{vrd}_{p,\rho}) \ \&\ \sigma(p) = v\right]
\end{array}
$$

Figure 4 Sequential Consistent Memory: Load, Store and Conditionals. (Side conditions included between brackets.)

The third line presents exactly the same operations, this time generated by instructions that operate on *volatile* references. Importantly, the action $\mathbf{vwr}_{\varrho,\mu}$ does not share the components $\mathcal{W}$, and $\mathcal{I}$ with its normal counterpart, a consequence of the fact that volatile variables only have SC semantics, and therefore will be processed directly from the memory.

Finally, the operations $\mathtt{lk}_\mu$ and $\mathtt{ul}_\mu$ are contributions of lock and unlock instructions on the lock value μ. In the dynamics of the program, μ needs to be a lock value (ℓ), or a placeholder value that will be eventually substituted by a lock value. The last operation, b, represents barrier instructions, and corresponds to the synchronization actions of Figure 2. With this definition of memory operations, we can precisely define temporary stores as being sequences of pairs of a thread identifier and a memory operation: $\delta \in (\mathcal{T}id \times \mathcal{O}pr)^*$.

The semantics of intra-thread (sequential) computation is given in the rest of Figure 3. At the bottom of Figure 3 we presents the definition of $[\![r]\!](\sigma, \delta, t)$, a function that takes as input a redex r, a store σ, a temporary store δ, and a thread id t returning a pair containing: (1) the memory operation to be added to the temporary store (where τ represents a reduction with no memory operation), and (2) the continuation of the command to be executed. Note that the only place where this definition uses the store σ and the temporary store δ is in choosing a *fresh placeholder* value for reads and a *free location* for reference creation.

The rule Single-Thread Step at the top right of the figure, shows how each thread contributes to the temporary store in a full configuration. We take $(t, \mathtt{c}) \parallel \mathsf{T}$ to be the extension of T to $\mathsf{T}' = \mathsf{T}[t \leftarrow \mathtt{c}]$.

Before proceeding with the semantics of the memory system, we remark that for lack of space, we defer the treatment of locks to Appendix B of the extended version of the paper [22]. Importantly, the treatment of locks is similar to the treatment of volatile variables as hinted by Table 1. Therefore, most arguments that apply to a volatile load apply to a lock instruction, and similarly a volatile store with an unlock. We only consider locks in the paper in the tables, to show that their treatment corresponds with the respective volatile operations.[9]

Sequential Consistency

We illustrate how to encode sequential consistency (SC) using this semantics in Figure 4. We indicate between square brackets "[]" side conditions that apply to each of the rules on the right hand side of the figure. To describe the semantics, we define a conflict relation between memory operations which is necessary to describe when two memory operations are

[9] The treatment of locks can be found in the extended version of this paper [22].

(in-)dependent, and therefore cannot be reordered. For SC, the conflict relation includes all pairs.

$$\forall \mathsf{mo}, \mathsf{mo}', \ \mathsf{mo} \,\#_{\text{SC}}\, \mathsf{mo}'$$

The memory semantics uses a *commutativity predicate* between a temporary store and a pair of a thread and a memory operation. This predicate states when a memory operation can bypass the operations in the given temporary store. For SC, commutativity is characterized by the following requirement stating that actions of the same thread cannot bypass each other in the temporary store:

$$\delta \overset{\frown}{\text{SC}} (t, \mathsf{mo}) \iff \forall (t, \mathsf{mo}') \in \delta, \ \neg((t, \mathsf{mo}') \,\#_{\text{SC}} (t, \mathsf{mo}))$$

Other memory models will have different commutativity predicates, and we anticipate that only in the case of Power, this predicate will involve more than the simple conflict relation (#). Notice that since SC does not have write atomicity relaxations we have no rules that can extend the set $\mathcal{W}$ of writes, nor the set $\mathcal{I}$ of placeholders, which we omitted. It is not hard to see that this semantics imposes SC since all read operations are performed, in-order, from the store σ.

4 Cookbook Semantics

Table 3 captures the conflict relation induced by [15] (extending Table 1 with our memory operations) and we will use it to parameterize the semantics of the cookbook-high language. That table does not impose ordering constraints between two normal memory accesses (and indeed no memory barriers are systematically added between these), which are then considered to be as relaxed as the target architecture allows for normal memory accesses. Of all the target architectures that we consider in this work, the weakest is Power. We will then assume that at the cookbook-high semantics, the behaviors allowed by Power memory operations are propagated for memory operations on normal Java references. We notice that while it is possible to enforce a stricter semantics by adding fences in between normal memory accesses, this would likely severely impact the performance of even sequential programs, a clearly undesirable result.

The resulting semantics then adheres to Power behavior for normal memory accesses (cf. [8]), and SC for volatiles. This is what we refer to as cookbook-high, and it is what we use to prove our correctness results. We concede that a weaker semantics for non-volatile variables could be considered, as it is indeed the case in the JMM [17], at the expense of more complicated and subtle reasoning to prove soundness of low-level implementations.

In this work, we consider a strict interpretation of the rules of [15]. We use relational notation to capture the information found in these tables. We write $\mathsf{mo} \,\#_{\text{CH}}\, \mathsf{mo}'$ to signify that a pair of memory operations mo and mo' have a conflict, if and mo defines a row and mo' defines a column, and the matching entry in the table has a conflict symbol #, or the condition in the entry is met by the memory operations (up to the obvious substitutions of formal parameters; for example $\mathsf{wr}_p \,\#_{\text{CH}}\, \mathsf{wr}_p$). This conflict relation will be used to know when two actions of the same thread can commute in the temporary store with each other, denoted $(t, \mathsf{mo}) \overset{\frown}{\text{CH}} (t, \mathsf{mo}')$.

However, the fact that this semantics has write-atomicity relaxations as explained before implies that to preserve a consistent semantics we need to impose constraints between different threads on the commutativity of operations. For instance, a read action that sees a write before the latter has been made visible to all threads – a *read-early* action – cannot be

Table 3 Cookbook-High Conflict Relation ($\#_{\mathrm{CH}}$).

1st\2nd	$\mathsf{rd}_{p'}$	$\mathsf{rd}_{\rho'}$	$\overline{\mathsf{rd}_{\rho'}}$	$\mathsf{wr}_{p'}$	$\mathsf{wr}_{\rho'}$	$\mathbf{vrd}_{\mathsf{p}'}$	$\overline{\mathbf{vrd}_{\mathsf{p}'}}$	$\mathbf{vwr}_{\mathsf{p}'}$	$\mathtt{lk}_{\ell'}$	$\mathtt{ul}_{\ell'}$
rd_p				$p = p'$	#			#		#
rd_ρ				#	#			#		#
$\overline{\mathsf{rd}_\rho}$								#		#
$\mathsf{wr}_{\overline{p}}^{_,\mathcal{I}}$	$p = p'$	#	$\rho' \in \mathcal{I}$	$p = p'$	#			#		#
wr_ρ	#	#	#	#	#			#		#
$\mathbf{vrd}_{\mathsf{p}}$	#	#	#	#	#	#	#	#	#	#
$\overline{\mathbf{vrd}_{\mathsf{p}}}$	#	#	#	#	#	#	#	#	#	#
$\mathbf{vwr}_{\mathsf{p}}$						#	#	#	#	#
$\mathtt{lk}_\ell$	#	#	#	#	#	#	#	#	#	#
$\mathtt{ul}_\ell$						#	#	#	#	#

accepted as performed by the issuing thread – *committed* – before the write is performed made visible to all threads. Hence, we overload the conflict relation ($\#$) to pairs of a thread and a memory operation. The minimal requirements for this relation are presented using the notation $\#_\blacktriangleleft$:

$$\varrho = \varrho' \text{ or } \varrho \in \mathcal{P}\mathit{lHold} \text{ and } t' \in \mathcal{W} \text{ or } \mathcal{I} \neq \emptyset \neq \mathcal{I}' \Rightarrow \begin{array}{c} (t, \mathsf{wr}_\varrho^{\mathcal{W},\mathcal{I}}) \;\#_\blacktriangleleft\; (t', \mathsf{wr}_\varrho^{\mathcal{W}',\mathcal{I}'}) \\ (t, \mathsf{wr}_\varrho^{\mathcal{W},\mathcal{I}}) \;\#_\blacktriangleleft\; (t', \mathsf{rd}_{\varrho'}) \end{array} \tag{2}$$

$$(t, \mathsf{wr}^{\mathcal{W},\mathcal{I}\cup\{\rho\}}) \;\#_\blacktriangleleft\; (t', \overline{\mathsf{rd}_\rho}) \tag{3}$$

$$\varrho = \varrho' \text{ or } \varrho \in \mathcal{P}\mathit{lHold} \Rightarrow (t, \mathsf{rd}_\varrho) \;\#_\blacktriangleleft\; (t, \mathsf{wr}_{\varrho'}) \tag{4}$$

$$\varrho = \varrho' \text{ or } \varrho \in \mathcal{P}\mathit{lHold} \text{ and } t' \in \mathcal{W} \cup \{t\} \Rightarrow (t, \mathsf{wr}_\varrho^{\mathcal{W}}) \;\#_\blacktriangleleft\; (t', \mathsf{wr}_{\varrho'}) \tag{5}$$

$$\begin{array}{c} (t, \overline{\mathsf{rd}_\rho}) \;\#_\blacktriangleleft\; (t, \mathsf{t}_{\overline{\rho}}) \text{ and } (t, \overline{\mathsf{rd}_\rho}) \;\#_\blacktriangleleft\; (t, \mathsf{e}_{\overline{\rho}}) \\ (t, \mathsf{t}_{\overline{\rho}}) \;\#_\blacktriangleleft\; (t, \mathsf{wr}) \text{ and } (t, \mathsf{e}_{\overline{\rho}}) \;\#_\blacktriangleleft\; (t, \mathsf{wr}) \end{array} \tag{6}$$

Condition (2) implies the obvious data dependencies between a write action and a subsequent action on the same reference by the same thread. Notice that a placeholder can potentially represent any reference, and hence, when the target of a write is undefined, any action by the same thread potentially conflicts with it. Moreover, the condition states that if a write action by t has been made visible to t', then this write action will conflict with subsequent actions on the same reference (with a conservative over-approximation for placeholders) by thread t'. Finally, it establishes that a write that has been seen early ($\mathcal{I} \neq \emptyset$) conflicts with any other action on the same reference (cf. placeholder), either after or before ($\mathcal{I}' \neq \emptyset$). Condition 3 establishes a natural constraint between an early write, and any early read that used this write. Condition 4 is similar to the first constraint established by 2, except that it operates on a read followed by a write of the same thread. Condition 5 requires that writes that potentially target the same reference be kept in order if the first has been made visible to the thread issuing the second. Finally, condition 6 requires that a speculation action be ordered w.r.t. the read action that originated the value of the conditional; and that write actions (following [24]) cannot bypass prior speculative branching actions. Since volatile references are not subject to write-atomicity relaxations, their constraints are fully defined by Table 3. In particular, many of the constraints in $\#_\blacktriangleleft$ take a conservative approach when relating placeholder values, whose target references are not known (e.g. Equation 2). However, when relating a normal memory access and a volatile access in cookbook-high, even if one or both

(SC) Speculation Commit

$$(\sigma, (t, \mathsf{mo}) \cdot \delta, \mathsf{T}) \underset{\text{CH}}{\hookrightarrow} (\sigma, \delta, \mathsf{T}) \qquad \left[\mathsf{mo} \in \{\mathtt{t}_\rho^{tt}, \mathtt{e}_\rho^{ff}\}\right]$$

(VW) Volatile Write

$$(\sigma, \delta_0 \cdot (t, \mathbf{vwr}_{\mathsf{p},v}) \cdot \delta_1, \mathsf{T}) \underset{\text{CH}}{\hookrightarrow} (\sigma[\mathsf{p} \leftarrow v], \delta_0 \cdot \delta_1, \mathsf{T}) \qquad \left[\delta_0 \overset{\frown}{\text{CH}} (t, \mathbf{vwr}_{\mathsf{p},v})\right]$$

(VR) Volatile Read

$$(\sigma, \delta_0 \cdot (t, \mathbf{vrd}_{\mathsf{p},\rho}) \cdot \delta_1, \mathsf{T}) \underset{\text{CH}}{\hookrightarrow} (\sigma, \delta_0 \cdot (t, \overline{\mathbf{vrd}_{\rho,v}}) \cdot (\delta_1[\rho \leftarrow v]), \mathsf{T}[\rho \leftarrow v]) \qquad \left[\begin{array}{c} \sigma(\mathsf{p}) = v \\ \delta_0 \overset{\frown}{\text{CH}} (t, \mathbf{vrd}_{\mathsf{p},\rho}) \end{array}\right]$$

(VRC) Volatile Read Commit

$$(\sigma, \delta_0 \cdot (t, \overline{\mathbf{vrd}_{\rho,v}}) \cdot \delta_1, \mathsf{T}) \underset{\text{CH}}{\hookrightarrow} (\sigma, \delta_0 \cdot \delta_1, \mathsf{T}) \qquad \left[\delta_0 \overset{\frown}{\text{CH}} (t, \mathbf{vrd}_{\rho,v})\right]$$

(NW) Normal Write

$$(\sigma, \delta_0 \cdot (t, \mathsf{wr}_{p,v}^{\mathcal{W},\mathcal{I}}) \cdot \delta_1, \mathsf{T}) \underset{\text{MM}}{\hookrightarrow} (\sigma[p \leftarrow v], \delta_0 \cdot \delta_1, \mathsf{T}) \qquad \left[\begin{array}{c} \delta_0 \overset{\frown}{\text{MM}} (t, \mathsf{wr}_{p,v}^{\mathcal{W},\mathcal{I}}) \\ \text{MM} \in \{\text{CH}, \text{CL}, \text{PPC}, \text{TSO}\} \end{array}\right]$$

(NR) Normal Read

$$(\sigma, \delta_0 \cdot (t, \mathsf{rd}_{p,\rho}) \cdot \delta_1, \mathsf{T}) \underset{\text{MM}}{\hookrightarrow} (\sigma, \delta_0 \cdot (\delta_1[\rho \leftarrow v]), \mathsf{T}[\rho \leftarrow v]) \qquad \left[\begin{array}{c} \sigma(p) = v \\ \delta_0 \overset{\frown}{\text{MM}} (t, \mathsf{rd}_{p,\rho}) \\ \text{MM} \in \{\text{CH}, \text{CL}, \text{PPC}, \text{TSO}\} \end{array}\right]$$

(EW) Write Early

$$(\sigma, \delta_0 \cdot (t, \mathsf{wr}_{\varrho,v}^{\mathcal{W},\mathcal{I}}) \cdot \delta_1, \mathsf{T}) \underset{\text{MM}}{\hookrightarrow} (\sigma, \delta_0 \cdot (t, \mathsf{wr}_{\varrho,v}^{\mathcal{W}',\mathcal{I}}) \cdot \delta_1, \mathsf{T}) \qquad \left[\begin{array}{c} \mathcal{W} \subset \mathcal{W}' \\ \text{MM} \in \{\text{CH}, \text{CL}, \text{PPC}, \text{TSO}\} \end{array}\right]$$

(ER) Read Early

$$(\sigma, \delta_0 \cdot (t, \mathsf{wr}_{\varrho,\mu}^{\mathcal{W},\mathcal{I}}) \cdot \delta_1 \cdot (t', \mathsf{rd}_{\varrho,\rho}) \cdot \delta_2, \mathsf{T}) \underset{\text{MM}}{\hookrightarrow}$$
$$(\sigma, \delta_0 \cdot (t, \mathsf{wr}_{\varrho,\mu}^{\mathcal{W},\mathcal{I}\cup\{\rho\}}) \cdot \delta_1 \cdot (t', \overline{\mathsf{rd}_{\rho,\mu}}) \cdot (\delta_2[\rho \leftarrow \mu]), \mathsf{T}[\rho \leftarrow \mu]) \qquad \left[\begin{array}{c} t' \in \mathcal{W} \\ \delta_1 \overset{\frown}{\text{MM}} (t', \mathsf{rd}_{\varrho,\rho}) \\ \delta_0 \ \overset{\frown}{\text{MM}\mathcal{S}ync} \ (t', \mathsf{rd}_{\varrho,\rho}) \\ \text{MM} \in \{\text{CH}, \text{CL}, \text{PPC}, \text{TSO}\} \end{array}\right]$$

(ERC) Commit Read Early

$$(\sigma, \delta_0 \cdot (t, \overline{\mathsf{rd}_{\rho,v}}) \cdot \delta_1, \mathsf{T}) \underset{\text{MM}}{\hookrightarrow} (\sigma, \delta_0 \cdot \delta_1, \mathsf{T}) \qquad \left[\begin{array}{c} \delta_0 \overset{\frown}{\text{MM}} (t, \overline{\mathsf{rd}_{\rho,v}}) \\ \text{MM} \in \{\text{CH}, \text{CL}, \text{PPC}, \text{TSO}\} \end{array}\right]$$

(FN) Fence

$$(\sigma, \delta_0 \cdot (t, b) \cdot \delta_1, \mathsf{T}) \underset{\text{MM}}{\hookrightarrow} (\sigma, \delta_0 \cdot \delta_1, \mathsf{T}) \qquad \left[\begin{array}{c} \delta_0 \overset{\frown}{\text{MM}} (t, b) \\ \text{MM} \in \{\text{CL}, \text{PPC}, \text{TSO}\} \end{array}\right]$$

Figure 5 A high-level semantics for the Java cookbook (induced by Power).

of them have placeholder values we know that they could not conflict since $\mathcal{P}tr \cap \mathcal{V}ptr = \emptyset$. Therefore, all the conditions requiring $\varrho = \varrho'$ or $\varrho \in \mathcal{P}l\mathcal{H}old$ as a precondition do not apply if one memory access is volatile and the other is not. The commutativity predicate of the cookbook-high language is thus:

$$\delta \overset{\frown}{\text{CH}} (t, \mathsf{mo}) \iff \forall\ (t', \mathsf{mo}') \in \delta,\ \neg(t', \mathsf{mo}')\ \#_{\blacktriangleleft}\ (t, \mathsf{mo}) \text{ and } t = t' \Rightarrow \neg(\mathsf{mo}' \#_{\text{CH}} \mathsf{mo})$$

Figure 5 shows the rules capturing memory model behavior for cookbook-high. These rules are only concerned with the memory, and follow a judgment of the form

$$(\sigma, \delta, \mathsf{T}) \underset{\text{CH}}{\hookrightarrow} (\sigma', \delta', \mathsf{T}')$$

where the arrow is annotated with the memory model relation (defining a corresponding commutativity) being considered. Notice that many of the rules in Figure 5 concern multiple memory models. Now we are only concerned with the ones that refer to CH (i.e. all but (FN)). The rule (SC) simply establishes that a conditional speculation operation, whose condition has been validated (i.e. $\mathtt{t}_\rho^{tt}$ or $\mathtt{e}_\rho^{ff}$) can be removed from the temporary store when it reaches the front. Note that the placeholder substitution operation does not eliminate

the placeholder (i.e. $\mathtt{t}_\rho[\rho \leftarrow v] = \mathtt{t}^v_\rho$ as opposed to $\mathtt{t}_v$) since we need to keep record of the placeholder value for the definition of commutativity. The rule (VW) and which refers to a volatile write is identical to the write rule for SC of Figure 4. It reflects the fact that volatile writes have SC semantics in cookbook-high. The rules (VR) and (VRC), which can always be applied immediately one after the other, mimic the read rule of Figure 4, again reflecting the fact that volatile reads are always satisfied from the memory, and have SC semantics. However, unlike in Figure 4 the (VR) rule does not directly remove the read operation $(t, \mathbf{vrd}_{\mathsf{p},\rho})$, instead replacing it with the "read mark" memory operation $(t, \overline{\mathbf{vrd}_{\rho,v}})$ where v is the current value of p in the store. This read mark memory operation can then (perhaps immediately) be removed by rule (VRC) to mimic the SC read rule. Thus, the read mark operation serves as a marker indicating that a read operation, say $\mathbf{vrd}_{\mathsf{p},\rho}$ has been performed (i.e. the value has been queried from the memory), and the placeholder ρ has been substituted by the value v, but the operation has not yet been removed from the temporary store, instead simply replaced for $\overline{\mathbf{vrd}_{\rho,v}}$ which through the commutativity predicate might limit the applicability of subsequent memory operations in the temporary store. Clearly, this marker is unnecessary for volatile references. We include it to simplify the simulation argument between cookbook-high and cookbook-low (presented in the next section). On the contrary, for normal references $\overline{\mathsf{rd}_{\rho,v}}$ is used to limit the commutativity of subsequent operations.[10]

Rules(NW) and (NR) represent the commitment of a normal reference write and read, resp. These rules resemble their SC counterparts, except that the write operation $\mathsf{wr}^{\mathcal{W},\mathcal{I}}_{p,v}$ might have been propagated to other threads (the threads in the $\mathcal{W}$ set) and it might have already been used to satisfy some reads in the temporary store early (the reads whose placeholders are in $\mathcal{I}$). Otherwise this rule is identical to the SC version up to the new definition of the commutativity predicate ($\widehat{\textsc{ch}}$). Similarly, a read that has not yet been performed (through a read-early action (ER)) can be performed, querying the store, as long as it commutes with all other operations in the temporary store before it.

Perhaps the most interesting rules are (EW) and (ER) for early-writes and early-reads. Recall that a write can be prematurely propagated to certain threads (i.e. before reaching the store). The component $\mathcal{W}$ of a memory write operation is a set of thread IDs that can immediately see this write in the temporary store. The (EW) rule extends the set to new threads. To be able to use these writes that are propagated through the temporary store, the rule (ER) can reduce a read action $\mathsf{rd}_{\varrho,\rho}$ by substituting the placeholders ρ that it generated by the value (or placeholder) written by the propagated write. As it is the case in (VR) the action does not immediately disappear from the temporary store, but a marker is added, with the placeholder that was substituted and the value (or placeholder) that it was substituted with. This marker is important because a read action can prevent the commutativity of subsequent operations, and if we were to simply remove it we lose that information. Moreover we note that the placeholder of the original read (ρ) is added to the set $\mathcal{I}$ of reads that were fulfilled early by the matching write. This again, is to preserve commutativity constraints as discussed above. To end this case we notice that one of the side-conditions of (ER) requires that $\delta_0 \ \widehat{{}_{\textsc{mm}}\mathcal{S}ync}\ (t', \mathsf{rd}_{\varrho,\rho})$ which considers only the synchronization operations of δ_0 and for $\textsc{ch}$ is defined as:

$$\delta_0 \ \widehat{\textsc{ch}_{\mathcal{S}ync}}\ (t', \mathsf{rd}_{\varrho,\rho}) \iff \forall (t', \mathsf{mo}) \in \delta_0, \neg(\mathsf{mo} \in \{\mathbf{vrd}, \overline{\mathbf{vrd}}, \mathtt{lk}_\ell\})$$

Thus, the read action $(t', \mathsf{rd}_{\varrho,\rho})$ should not bypass synchronization actions by t' in δ_0 (i.e.,

[10] In this sense, it plays a fundamental role in the definition of Power barriers.

in the temporary store before the write). This is because synchronization actions in δ_0 performed by t' could prevent the execution of the read (for example a pending volatile read by t'). The final rule (ERC) is similar to (NR) and serves to eliminate read markers from the temporary store.

The final judgment below is the obvious composition between the intra-thread semantics of Figure 3 and the memory semantics defined in this section.

$$\textsc{Composed Semantics}\quad \frac{(\sigma,\delta,\mathsf{T}) \xrightarrow[t]{\mathsf{mo}} (\sigma',\delta',\mathsf{T}') \quad \text{or} \quad (\sigma,\delta,\mathsf{T}) \underset{\text{MM}}{\hookrightarrow} (\sigma',\delta',\mathsf{T}')}{(\sigma,\delta,\mathsf{T}) \overset{\text{MM}}{\Longrightarrow} (\sigma',\delta',\mathsf{T}')}$$

As an example, let us reconsider the program motivating store-atomicity that we saw in section 2 in the syntax of cookbook-high where we use tuples as possible values.

$$p := p' \quad \| \quad \texttt{let } x = !p \texttt{ in } x := p \quad \| \quad \begin{array}{l}\texttt{let } x = !p' \texttt{ in}\\ \texttt{let } y = !x \texttt{ in } (x,y)\end{array}$$

If we use thread names t_0, t_1 and t_2 for these threads we observe that by executing them in order we can reach a configuration with a temporary store of the form

$$(t_0,\mathsf{wr}_{p,p'})\cdot(t_1,\mathsf{rd}_{p,\rho})\cdot(t_1,\mathsf{wr}_{\rho,p'})\cdot(t_2,\mathsf{rd}_{p',\rho'})\cdot(t_2,\mathsf{rd}_{\rho',\rho''})$$

Then by a (EW) rule in the write of t_0 we can extend the visibility to t_1, and use (ER) in the read by t_1 obtaining

$$(t_0,\mathsf{wr}^{\{t_0,t_1\},\{\rho\}}_{p,p'})\cdot(t_1,\overline{\mathsf{rd}_{\rho,p'}})\cdot(t_1,\mathsf{wr}_{p',p})\cdot(t_2,\mathsf{rd}_{p',\rho'})\cdot(t_2,\mathsf{rd}_{\rho',\rho''})$$

We can now repeat these steps for the write of t_1 extending their visibility to t_2.

$$(t_0,\mathsf{wr}^{\{t_0,t_1\},\{\rho\}}_{p,p'})\cdot(t_1,\overline{\mathsf{rd}_{\rho,p'}})\cdot(t_1,\mathsf{wr}^{\{t_1,t_2\},\{\rho'\}}_{p',p})\cdot(t_2,\overline{\mathsf{rd}_{\rho',p}})\cdot(t_2,\mathsf{rd}_{p,\rho''})$$

And now we can see that the last read $(t_2,\mathsf{rd}_{p,\rho''})$ can proceed with a (NR) rule, and since the write $(t_0,\mathsf{wr}^{\{t_0,t_1\},\{\rho\}}_{p,p'})$ has not been made visible to t_2, this read will return the default value in memory (assumed to be NULL). This is a behavior that is possible in Power, and we propagate for normal variables in cookbook-high.

JMM Guarantees

The semantics of the JMM [17, 19] is defined in terms of a *justification procedure* which commits actions of a hypothetical execution one by one. This semantic style is very different from the operational one introduced in this section. However, we have proved (in Appendix C of the extended version of this paper [22]) that every execution of cookbook-high corresponds to a legal execution of the JMM as redefined by [19].

▶ **Theorem 1.** *All the executions of cookbook-high as per the semantics presented in this section can be justified as per the formalization of the* JMM *of [19].*

Proof. (Sketch) Space limitations preclude us from presenting the formal definition of the JMM as presented in [19].[11] While the semantics of Cookbook-High is operational, and events are generated as the program is executed, the semantics of the JMM is axiomatic. In

[11] The definition and full proof can be found in the appendix of the extended version of this paper [22].

Table 4 Cookbook-Low Conflict Relation ($\#_{\text{CL}}$).

1st\2nd	$\mathsf{rd}_{p'}$	$\mathsf{rd}_{\rho'}$	$\overline{\mathsf{rd}}_{\rho'}$	$\mathsf{wr}_{p'}$	$\mathsf{wr}_{\rho'}$
rd_p				$p = p'$	$\#$
rd_ρ				$\#$	$\#$
$\overline{\mathsf{rd}}_\rho$					
$\mathsf{wr}_{\overline{p}}^{-,\mathcal{I}}$	$p = p'$	$\#$	$\rho' \in \mathcal{I}$	$p = p'$	$\#$
wr_ρ	$\#$	$\#$	$\#$	$\#$	$\#$

$$\mathsf{mo} \in \{\mathsf{rd}, \overline{\mathsf{rd}}\} \Rightarrow \begin{cases} \mathsf{mo} \ \# \ \langle \mathtt{ld} | _ \rangle \ \& \\ \langle _ | \mathtt{ld} \rangle \ \# \ \mathsf{mo} \end{cases}$$

$$\mathsf{wr} \ \# \ \langle \mathtt{st} | _ \rangle \ \& \ \langle _ | \mathtt{st} \rangle \ \# \ \mathsf{wr}$$

$$b \ \# \ b'$$

the JMM, assuming a hypothetical execution ξ one justifies the execution with a series of steps that – using other executions – justify each of the actions in ξ. Our proof consists of a process to justify a final execution ξ, but in our treatment, the final execution is not known ahead of time. Instead, we justify steps as they happen, and show that if the operational semantics of Cookbook-High makes progress, all of the generated steps can be justified. We show that axioms that each time a step of Cookbook-High happens, it can be committed by the axioms of the JMM, meaning that all the Cookbook-High actions are permisible actions of the JMM. This argument extended to full traces guarantees the statement of the theorem. ◀

This result is not surprising since the semantics of cookbook-high is much more restrictive than the intended semantics of the JMM. As a corollary we obtain that the guarantees that are respected by the JMM [5, 19] also hold for us.

▶ **Corollary 2** (JMM properties). *The following properties hold for Cookbook-High:*

- *Cookbook-High respects the DRF guarantee.*
- *Cookbook-High prevents* out-of-thin-air *reads.*
- *The projection of the semantics of cookbook-high to volatile variables and locks respects the* SC *semantics.*

Proof. (Sketch) This proof is an immediate consequence of the previous theorem, and the fact that all of these properties hold for the JMM [19]. ◀

This is a consequence of the theorem above and [19]. Moreover, cookbook-high provides SC semantics for volatile variables and locks (with respect to each other). The results above are some fundamental requirements that the JMM should satisfy [17]. We emphasize that the non-trivial proof of the theorem above can be found in Appendix C of the extended version of the paper [22].

5 Cookbook-Low: Definition, Compilation, Simulation

We now present the intermediate representation of [15] as the cookbook-low language of Figure 2. Our first result is that the semantics of cookbook-low simulates the semantics of cookbook-high if the rules presented in Table 5 are respected when compiling from cookbook-high to cookbook-low. These rules indicate that in-between any two memory accesses by the same thread at the cookbook-high level, there must be a barrier between the corresponding memory accesses in the cookbook-low level as indicated by that cell in the table mediating them. We omit the reference name, and other parameters of memory operations since they are unnecessary. Similarly, operations $\overline{\mathsf{rd}}$ and $\overline{\mathbf{vrd}}$, which are not directly emitted by threads are omitted, but their constraints are similar to those of rd and $\mathbf{vrd}$, resp.

The main differences between cookbook-high and cookbook-low are that:

Table 5 Cookbook-High to Cookbook-Low Barriers (H-L).

1st\2nd	rd	wr	vrd	vwr	lk$_{\ell'}$	ul$_{\ell'}$
rd				⟨l|s⟩		⟨l|s⟩
wr				⟨s|s⟩		⟨s|s⟩
vrd	⟨l|l⟩	⟨l|s⟩	⟨l|l⟩	⟨l|s⟩	⟨l|l⟩	⟨l|s⟩
vwr			⟨s|l⟩	⟨s|s⟩	⟨s|l⟩	⟨s|s⟩
lk$_\ell$	⟨l|l⟩	⟨l|s⟩	⟨l|l⟩	⟨l|s⟩	⟨l|l⟩	⟨l|s⟩
ul$_\ell$			⟨s|l⟩	⟨s|s⟩	⟨s|l⟩	⟨s|s⟩

1. In cookbook-low there are no *volatile* references. Hence, we collapse the set $\mathcal{V}ptr$ of cookbook-high into $\mathcal{P}tr$. If $\mathcal{P}tr_H$ denotes the set of normal references at the cookbook-high level, and $\mathcal{P}tr_L$ the set of normal references at cookbook-high, then $\mathcal{V}ptr \cup \mathcal{P}tr_H \subseteq \mathcal{P}tr_L$. Recall cookbook-high require that $\mathcal{P}tr_H \cap \mathcal{V}ptr = \emptyset$, and therefore in cookbook-low there should be no confusion when considering $\mathcal{V}ptr$ as being part of $\mathcal{P}tr_L$.
2. The cookbook-low semantics includes barrier operations defined by $\mathcal{S}ync$CBL as given in Figure 2. These barriers impose the obvious conflict relation with respect to other memory accesses of the same thread.

The semantics of the memory component of cookbook-low uses the conflict relation $\#_{\text{CL}}$ of Table 4 to define commutativity. All the rules of Figure 5 not involving volatile references (that is (VW), (VR) and (VRC)) apply to cookbook-low. The only rule that was not explained in the previous section is (FN), which simply dictates when a barrier operation can be removed from the temporary store.

5.1 Compilation

Following [15] we specify sufficient conditions to enforce the cookbook-high semantics, which in turn is a conservative approximation of the JMM. Table 5 indicates for each cell which barrier – if any – has to be inserted in between the cookbook-low memory accesses that implement the corresponding cookbook-high memory accesses. We use Table 5 as a function with two arguments corresponding to the first and second memory operation, with the result depicted within the corresponding cell, which we will write as H-L $(\mathsf{mo}, \mathsf{mo}') = b$. Therefore we write H-L $(\mathsf{mo}, \mathsf{mo}') = b$ if the cell in row mo and column mo' contains a barrier b.

Below we present an intuitive statement of our main theorem relating programs of Cookbook-High and Cookbook-Low. The formal statement, and its proof are relegated to the Appendix A of the extended version of this paper [22].

▶ **Theorem 3** (Cookbook-Low simulates Cookbook-High). *Given thread systems, T_H of Cookbook-High and T_L of Cookbook-Low related by related by the function induced by Table 5. Each time that a thread in the system T_L can take a step by the composed semantic, the thread system T_H can take a similar step leading to related configurations.*

Proof Sketch. The proof is based on the construction of a simulation relation between configurations of Cookbook-High and Cookbook-Low. In a nutshell, normal variables in Cookbook-High correspond to identical variables in Cookbook-Low, whereas volatile variables in Cookbook-High are mapped into normal variables of Cookbook-Low. The stores of both configurations are the same, and the temporary stores are strongly related, where the relation takes into account the effects that the barriers imposed by Table 5 have on the shape of

$$\begin{array}{lcl} \texttt{x86}(\langle \texttt{l}|\texttt{l}\rangle) & = & \texttt{skip} \\ \texttt{x86}(\langle \texttt{l}|\texttt{s}\rangle) & = & \texttt{skip} \end{array} \qquad \begin{array}{lcl} \texttt{x86}(\langle \texttt{s}|\texttt{s}\rangle) & = & \texttt{skip} \\ \texttt{x86}(\langle \texttt{s}|\texttt{l}\rangle) & = & \mathsf{mfence} \end{array}$$

Figure 6 x86 Compilation Strategy.

$$\begin{array}{lcl} \texttt{PPC}(\langle \texttt{l}|\texttt{l}\rangle) & = & \mathsf{sync} \\ \texttt{PPC}(\langle \texttt{l}|\texttt{s}\rangle) & = & \mathsf{lwsync} \end{array} \qquad \begin{array}{lcl} \texttt{PPC}(\langle \texttt{s}|\texttt{s}\rangle) & = & \mathsf{lwsync} \\ \texttt{PPC}(\langle \texttt{s}|\texttt{l}\rangle) & = & \mathsf{sync} \end{array}$$

Figure 7 PPC Compilation Strategy.

the Cookbook-Low temporary store w.r.t. the shape of the corresponding Cookbook-High temporary store. Finally, the program code of Cookbook-High and Cookbook-Low are related by an auxiliary *well-compiled* relation which enforces that fences have been added to the Cookbook-Low programs in accordance with Table 5.

As in any backward simulation, the proof strategy consists in a case analysis of all the possible Cookbook-Low step, showing that starting in related Cookbook-Low and Cookbook-High configurations, a new Cookbook-High configuration can be reached by executing zero or more steps in the Cookbook-High semantics. ◀

This means that any behavior produced by the Cookbook-Low configuration can also be produced by the Cookbook-High configuration, proving that Table 5 induces a correct compilation from Cookbook-High to Cookbook-Low.

6 x86 and Power Simulation

In this section, we present the semantics of the x86 and Power architectures in our core language. These definitions are inspired by [21, 24].

6.1 x86

To define the semantics of x86 it suffices to consider Figure 5 where the commutativity relation variable $\overset{\frown}{\text{MM}}$ is instantiated with $\overset{\frown}{\text{TSO}}$ induced by the conflict relation $\#_{\text{TSO}}$ below.

$$\mathsf{mo} \ \#_{\text{TSO}} \ \mathsf{mo}' \iff (\mathsf{mo} \neq \mathsf{wr}) \ \vee \ (\mathsf{mo} \neq \overline{\mathsf{rd}}) \ \vee \ (\mathsf{mo} = \mathsf{wr}_p \ \wedge \ \mathsf{mo}' \neq \mathsf{rd}_p)$$

Moreover, we require that the write-early rule (EW) cannot propagate a write $(t, \mathsf{wr}_\varrho^{\mathcal{W},\mathcal{I}})$ to a set $\mathcal{W}$ larger than $\{t\}$. In other words, writes can only be read early by the thread that emitted them (in essence modeling the store-buffers of TSO [21]). Finally, we have only one barrier instruction that imposes the following orderings.

$$\mathsf{wr} \ \#_{\text{TSO}} \ \mathsf{mfence} \qquad \mathsf{mfence} \ \#_{\text{TSO}} \ \mathsf{rd} \qquad \mathsf{mfence} \ \#_{\text{TSO}} \ \mathsf{mfence}$$

The implementation of the cookbook-low barrier operations in x86 is given in Figure 6. We say that an x86 command c_x is well-compiled w.r.t. to a cookbook-low source command c_L if c_x is obtained from c_L by substituting all the barrier operations according to Figure 6. A similar definition relates a x86 temporary store δ_x and a cookbook-low temporary store δ_L.

▶ **Theorem 4** (Cookbook-Low to x86 Simulation). *Given a thread system T_L in cookbook-low, an x86 thread system obtained from T_L by substituting the barriers according to Figure 6. This substitution establishes a simulation between their x86 and cookbook-low semantics.*[12]

[12] The proofs of this section are embedded in the text in the extended version [22].

In combination with Theorem 1 and Theorem 3 we obtain an end to end argument for the rules of the cookbook.

6.2 Power

Power is the weakest architecture considered in this paper, and the motivation for the write atomicity relaxation in the cookbook-high language semantics. The semantics of Power is defined in Figure 5 where we instantiate the memory model to $\textsc{ppc}$ (i.e. we consider the $\overset{\frown}{\textsc{ppc}}$ reordering relation).

We allow all normal memory operations of the same thread to commute in Power as long as they respect the constraints imposed by the minimal conflict $\#_{\blacktriangleleft}$, with the additional constraint that read operations to the same reference cannot be reordered (i.e. $\mathsf{rd}_p \; \#_{\textsc{ppc}} \; \mathsf{rd}_p$). Unlike the barriers we have considered thus far, Power barriers can impose global constraints that order and add visibility restrictions on operations of different threads. This is referred to in [23] and in [24] as cumulativity effects.

We first introduce the constraints imposed on $\overset{\frown}{\textsc{ppc}}$ by the sync barrier of Power encoded in the conflict relation $\#_{\textsc{ppc}}$, where we implicitly assume that barrier operations conflict with each other.

$$(t, \mathsf{wr}) \; \#_{\textsc{ppc}} \; (t, \mathsf{sync}) \; \#_{\textsc{ppc}} \; (t, \mathsf{wr}) \qquad (7)$$

$$(t, \mathsf{rd}) \; \#_{\textsc{ppc}} \; (t, \mathsf{sync}) \; \#_{\textsc{ppc}} \; (t, \mathsf{rd}) \qquad (8)$$

$$(t, \overline{\mathsf{rd}}) \; \#_{\textsc{ppc}} \; (t, \mathsf{sync}) \; \#_{\textsc{ppc}} \; (t, \overline{\mathsf{rd}}) \qquad (9)$$

$$(t, \mathsf{wr}^{\mathcal{W} \cup \{t'\}, \mathcal{I}}) \; \#_{\textsc{ppc}} \; (t', \mathsf{sync}) \qquad (10)$$

Notice that Equation 9 imposes strong ordering constraints between sync operations and reads that perhaps have been performed early. More importantly, Equation 10 reflects a conflict between a write in thread t, which is visible to thread t', and a subsequent sync by t'. In this sense, sync is a very strong barrier, because it imposes ordering between any two operations of the same thread, and moreover, imposes ordering between the write operations that have been made visible to a thread, and the thread's own operations.

A much weaker barrier is lwsync, whose conflict and commutativity relations we define below. Unlike sync, the commutativity of lwsync might depend on a number of prior operations performed by the thread before (in particular w.r.t. the action $\overline{\mathsf{rd}}$ as we shall see). Therefore, the last rule below (14) is simply stated in terms of commutativity ($\overset{\frown}{\textsc{ppc}}$) instead of conflict ($\#_{\textsc{ppc}}$).

$$(t, \mathsf{wr}) \; \#_{\textsc{ppc}} \; (t, \mathsf{lwsync}) \; \#_{\textsc{ppc}} \; (t, \mathsf{wr}) \qquad (11)$$

$$(t, \mathsf{rd}) \; \#_{\textsc{ppc}} \; (t, \mathsf{lwsync}) \; \& \; (t, \overline{\mathsf{rd}}) \; \#_{\textsc{ppc}} \; (t, \mathsf{lwsync}) \qquad (12)$$

$$t = t' \text{ or } t' \in \mathcal{W} \Rightarrow (t, \mathsf{wr}^{\mathcal{W}, \mathcal{I}}) \; \#_{\textsc{ppc}} \; (t', \mathsf{lwsync}) \qquad (13)$$

$$\left.\begin{array}{r} \delta = \delta' \cdot (t', \mathsf{lwsync}) \cdot \delta_3 \; \& \\ \delta' = \delta_0 \cdot (t, \mathsf{wr}_{\varrho,\mu}^{\mathcal{W} \cup \{t'\}, \mathcal{I} \cup \{\rho'\}}) \cdot \delta_1 \cdot (t', \overline{\mathsf{rd}_{\rho',\mu}}) \cdot \delta_2 \; \& \\ \neg(\delta_0 \; \overset{\frown}{\textsc{ppc}} \; (t, \mathsf{wr}_{\varrho,\mu}^{\mathcal{W} \cup \{t'\}, \mathcal{I} \cup \{\rho'\}})) \end{array}\right\} \Rightarrow \neg(\delta \; \overset{\frown}{\textsc{ppc}} \; (t', \mathsf{rd}_{\varrho',\mu'})) \qquad (14)$$

Notice that lwsync *does* allow the commutativity of $(t, \mathsf{wr}_p) \cdot (t, \mathsf{lwsync}) \; \overset{\frown}{\textsc{ppc}} \; (t, \mathsf{rd}_{p'})$ if $p \neq p'$. In other words, it does not prevent write-read reorderings (which is a typical capability of the TSO memory model). Secondly, we remark that Equation 13 is slightly stronger than the formalization of lwsync proposed in [24] (we follow [8]). However, as has been argued in [8], the behaviors that are not considered by this strengthening of lwsync have not been observed in the actual machines as reported by [24, 8, 3]. Finally, and perhaps the most complicated rule is given in Equation 14. This rules relates the constraints of an lwsync in between a $(t', \overline{\mathsf{rd}})$ operation and a (t', rd) operation. The rule states that the second read action can

only commute with the preceding temporary store if the write that the early-read action saw (i.e. $(t, \mathsf{wr}_{\varrho,\mu}^{\mathcal{W}\cup\{t'\},\mathcal{I}\cup\{\rho'\}})$) is in condition to be immediately performed (the rule is stated as a contrapositive). It is precisely this behavior of lwsync that enables IRIW behavior even if lwsync barriers are present between the reads of the reader threads (see [8]).

We have presented the compilation rules for the barriers in cookbook-low shown in Figure 7. We notice that compared to [15] we have replaced the compilation of the barrier $\langle \mathtt{l}|\mathtt{l}\rangle$ from lwsync to sync. To see why this is necessary, consider the compiled version of the IRIW example, where all references are volatile. We obtain the following program.

$$\left[\ \mathsf{p} := tt\ \right] \ \|\ \left[\ \mathsf{p}' := tt\ \right] \ \|\ \begin{bmatrix} \mathtt{let}\ x = !p\ \mathtt{in} \\ \quad \mathsf{lwsync}\,; \\ \quad \mathtt{let}\ y = !p'\ \mathtt{in}\ (x,y) \end{bmatrix} \ \|\ \begin{bmatrix} \mathtt{let}\ x = !p'\ \mathtt{in} \\ \quad \mathsf{lwsync}\,; \\ \quad \mathtt{let}\ y = !p\ \mathtt{in}\ (x,y) \end{bmatrix}$$

This program produces the relaxed behavior resulting in (tt, ff) for both threads, assuming that initially we have that p and p' hold ff in the store. We have tested a litmus Java example program, which contains only volatiles, in a Power V7 machine, and had been able to reproduce the relaxed behavior, which is clearly unacceptable according to [17], since volatiles have SC semantics. Moreover, this is a *data-race-free* program that violates the *DRF-guarantee.*

▶ **Theorem 5** (Cookbook-Low to Power Simulation)**.** *Given a thread system T_L in cookbook-low, a Power thread system is obtained from T_L by substituting the barriers according to Figure 7. This substitution establishes a simulation between their Power and cookbook-low semantics.*

About Power's lwsync

The semantics of lwsync considered in [24] is slightly weaker that the one considered here. In particular, an example where these two differ is presented in [24] under the name R01, which uses lwsync. Under that weaker interpretation of lwsync, Figure 7 would have to compile $\langle \mathsf{s}|\mathsf{s}\rangle$ into a sync for Power instead of the weaker lwsync. Unsurprisingly, that is the recommended implementation of sequentially consistent loads and stores in [6]. We clarify that this is *only* required for the implementation of *volatile* memory accesses (which are a lot less prevalent than normal memory accesses). Therefore this is unlikely to degrade the performance dramatically, and we could adopt it as a safe default. Theorem 5 is evidently true under this modification. This may or may not be considered an error in [15], according to the interpretation of lwsync chosen. According to [24] it is; however, the relaxation that is source of the error has not been observed in practice [24], making hard to convince compiler writers that it is necessary.

About ARM

We are tempted to make the argument that ARM is similar to Power leveraging [24]. Unfortunately [3] has found [24] to be inaccurate w.r.t. ARM. In [3] a different model is proposed, but it is claimed that some current ARM architectures suffer from a bug, which simply stated allows reads on the same reference to be reordered. Moreover, the new ARMv8 relaxed memory model is not yet quite well understood (see [12]). We note however that most of the behaviors discussed in [3] w.r.t. ARM are sound for the JMM, and could easily be incorporated into cookbook-high without affecting our proofs. Moreover, the conservative strategy of [15], which compiles all barriers to the sync-like `dmb` is guaranteed to satisfy our simulations as shown in Theorem 5.

7 Related Work

Our work is closely based on the intuitions provided by [15], whose structure and rules we try to follow as close as possible. We only depart from the informal description of [15] to remediate errors. Similar to [15], [4] presents an operational definition of a low-level language agnostic memory model, describing how the model, equipped with a notion of store atomicity and permissible instruction reorderings, can be used to capture various kinds of weak memory behavior.

Also related to our development is [6], where a compilation strategy for C++11 to Power is defined and verified correct. Unlike [6], however, we do not attempt to provide a concrete compilation strategy, instead verifying the minimal conditions required for compiling to architectures considered in [15]. In particular, this means we that need a *lingua franca* to relate the JMM and the architectures: we use the cookbook-low language for this purpose. Moreover, the roach motel semantics of the JMM is a fundamental property that we preserve, whereas this is not a concern in [6].

Recent efforts consider program analyses to insert fences [25, 2] to guarantee SC. We do not consider implementing SC for Java here which would be prohibitively inefficient for architectures like Power; recent work [18, 26] argues that the cost of ensuring end-to-end SC may be modest, assuming particular non-standard hardware support. Other works consider the elimination of redundant fences in weak memory systems (e.g., TSO [28, 20]). Since [15] enforces a conservative implementation of the JMM, we believe the cookbook-low formalization could be a starting point to consider similar results for Java (as informally argued at the end of [15]).

8 Conclusion

We present the first formal study of the minimal conditions necessary to guarantee the correct compilation of the JMM in different architectures as advocated by [15]. In doing so, we identify errors in the recommended implementation of volatiles for Power, and we propose *provably correct* repairs. We also define the semantics of the cookbook-low language, which we propose as an upper bound on how strong a memory model for Java can be, while not needing additional synchronization for the implementation of normal variables. Our work thus puts the “cookbook for compiler writers” [15] on a sound formal footing, a much needed exercise considering the current ongoing conversations about repairing the JMM.

Acknowledgements. We would like to thank Luc Maranget who provided us with access to a Power 7 machine to conduct some of our experiments. We are also grateful to Peter Sewell and Doug Lea who provided insightful comments on an early draft of our work. Finally we thank the anonymous reviewers whose recommendations have improved the quality of the paper.

References

1 Sarita V. Adve and Mark D. Hill. Weak Ordering – A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture, (ISCA 1990), Seattle, WA, June 1990*, pages 2–14, 1990.

2 Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't Sit on the Fence – A Static Analysis Approach to Automatic Fence Insertion. In *Computer Aided Verification, (CAV 2014), Vienna, Austria, July 18-22, 2014. Proceedings*, pages 508–524, 2014.

3 Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7, 2014.

4 Arvind and Jan-Willem Maessen. Memory Model = Instruction Reordering + Store Atomicity. In *33rd International Symposium on Computer Architecture (ISCA 2006), June 17-21, 2006, Boston, MA, USA*, pages 29–40, 2006.

5 David Aspinall and Jaroslav Ševčík. Formalising Java's Data Race Free Guarantee. In *Theorem Proving in Higher Order Logics, 20th International Conference, (TPHOLs 2007), Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, pages 22–37, 2007.

6 Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 2012), Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 509–520, 2012.

7 Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. OCTET: Capturing and Controlling Cross-thread Dependences Efficiently. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, (OOPSLA 2013), Indianapolis, IN, USA, October 26-31, 2013*, pages 693–712, 2013.

8 Gérard Boudol, Gustavo Petri, and Bernard P. Serpette. Relaxed Operational Semantics of Concurrent Programming Languages. In *Proceedings Combined 19th International Workshop on Expressiveness in Concurrency and 9th Workshop on Structured Operational Semantics, (EXPRESS/SOS 2012), Newcastle upon Tyne, UK, September 3, 2012.*, pages 19–33, 2012.

9 Delphine Demange, Vincent Laporte, Lei Zhao, Suresh Jagannathan, David Pichardie, and Jan Vitek. Plan B: a buffered memory model for Java. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 2013), Rome, Italy – January 23–25, 2013*, pages 329–342, 2013.

10 Cormac Flanagan and Matthias Felleisen. The Semantics of Future and Its Use in Program Optimizations. In *The 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 1995), San Francisco, California, USA, January 23-25, 1995*, pages 209–220, 1995.

11 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation, (PLDI 1993), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247, 1993.

12 JMM Mailing list: Developing the JEP 188: Java Memory Model Update. `http://mail.openjdk.java.net/mailman/listinfo/jmm-dev`, 2014.

13 Java Memory Model and Thread Specification, 2004.

14 Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

15 Doug Lea. The JSR-133 Cookbook for Compiler Writers. `http://g.oswego.edu/dl/jmm/cookbook.html`.

16 Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An Axiomatic Memory Model for POWER Multiprocessors. In *Computer Aided Verification – 24th International Conference, (CAV 2012), Berkeley, CA, USA, July 7–13, 2012 Proceedings*, pages 495–512, 2012.

17 Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *Special POPL Issue (DRAFT)*, 2005.

18 Daniel Marino, Abhayendra Singh, Todd D. Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A Case for an SC-preserving Compiler. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI 2011), San Jose, CA, USA, June 4-8, 2011*, pages 199–210, 2011.

19 Jaroslav Ševčík and David Aspinall. On Validity of Program Transformations in the Java Memory Model. In *ECOOP 2008 – Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7–11, 2008, Proceedings*, pages 27–51, 2008.

20 Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM*, 60(3):22, 2013.

21 Scott Owens, Susmit Sarkar, and Peter Sewell. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics, 22nd International Conference, (TPHOLs 2009), Munich, Germany, August 17-20, 2009. Proceedings*, pages 391–407, 2009.

22 Gustavo Petri, Jan Vitek, and Suresh Jagannathan. Cooking the Books: Formalizing JMM Implementation Recipes (Extended Version), 2015. `https://www.cs.purdue.edu/homes/gpetri/publis/CtB-long.pdf`.

23 PowerPC ISA. Version 2.06 Revision B. IBM, 2010.

24 Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER Multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI 2011), San Jose, CA, USA, June 4-8, 2011*, pages 175–186, 2011.

25 Dennis Shasha and Marc Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

26 Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd D. Millstein, and Madanlal Musuvathi. End-to-end Sequential Consistency. In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*, pages 524–535, 2012.

27 SPARC Corporation. *The SPARC Architecture Manual (V. 9)*. Prentice-Hall, Inc., 1994.

28 Viktor Vafeiadis and Francesco Zappa Nardelli. Verifying Fence Elimination Optimisations. In *Static Analysis – 18th International Symposium, (SAS 2011), Venice, Italy, September 14–16, 2011. Proceedings*, pages 146–162, 2011.

Defining Correctness Conditions for Concurrent Objects in Multicore Architectures

Brijesh Dongol[1], John Derrick[2], Lindsay Groves[3], and Graeme Smith[4]

1 **Department of Computer Science, Brunel University, UK**
`Brijesh.Dongol@brunel.ac.uk`
2 **Department of Computer Science, University of Sheffield, UK**
`J.Derrick@dcs.shef.ac.uk`
3 **School of Engineering and Computer Science, Victoria University of Wellington, New Zealand**
`lindsay@ecs.vuw.ac.nz`
4 **School of Information Technology and Electrical Engineering, The University of Queensland, Australia**
`smith@itee.uq.edu.au`

Abstract

Correctness of concurrent objects is defined in terms of conditions that determine allowable relationships between histories of a concurrent object and those of the corresponding sequential object. Numerous correctness conditions have been proposed over the years, and more have been proposed recently as the algorithms implementing concurrent objects have been adapted to cope with multicore processors with relaxed memory architectures.

We present a formal framework for defining correctness conditions for multicore architectures, covering both standard conditions for totally ordered memory and newer conditions for relaxed memory, which allows them to be expressed in uniform manner, simplifying comparison. Our framework distinguishes between order and commitment properties, which in turn enables a hierarchy of correctness conditions to be established. We consider the Total Store Order (TSO) memory model in detail, formalise known conditions for TSO using our framework, and develop sequentially consistent variations of these. We present a work-stealing deque for TSO memory that is not linearizable, but is correct with respect to these new conditions. Using our framework, we identify a new non-blocking compositional condition, fence consistency, which lies between known conditions for TSO, and aims to capture the intention of a programmer-specified fence.

1998 ACM Subject Classification D.1.3 Concurrent Programming, D.2.4 Software/Program Verification, F.1.2 Concurrent Programming, F.3.1 Specifying and Verifying and Reasoning about Programs, H.2.4 Systems

Keywords and phrases Concurrent objects, correctness, relaxed memory, verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.470

1 Introduction

This paper studies correctness conditions for *concurrent objects*, i.e., objects consisting of operations acting on shared data that may be executed concurrently by multiple processes. Because the operation calls of concurrent objects may overlap (as opposed to occurring one after another), their correctness is judged using a *correctness condition*, which is a relation on the behaviours of the concurrent object and its sequential specification object, i.e., a

© Brijesh Dongol, John Derrick, Lindsay Groves, and Graeme Smith;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 470–494

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

correctness condition provides an answer to the question: In what sense does a concurrent object implement its sequential specification?

Correctness conditions for concurrent objects has been the subject of study for nearly three decades and numerous conditions have been proposed. Shavit makes the case that different correctness conditions are needed in different circumstances [31]; weaker conditions provide greater scope for optimisation, but fewer behavioural guarantees. One cannot however, continually weaken correctness conditions in search of greater performance. Programmers require strong correctness conditions that ensure an abstract specification object (whose behaviours are understandable) can be safely substituted by a concurrent object (which provides better performance) within the programs that use these objects. The existence of these two opposing goals has meant that the number of accepted correctness conditions has actually increased over time (e.g., [31, 25]), instead of being consolidated into a unified correctness notion.

Most correctness conditions, including linearizability [23], have been developed under the assumption that hardware ensures *totally ordered memory*,[1] where reads and writes within a process are guaranteed to be executed in program order. Due to their use of local buffers, modern multicore architectures are not totally ordered, and only provide relaxed memory guarantees [1, 33], meaning memory instructions may be executed in a different order to that specified by the program. Such reorderings can be avoided by introducing **`fence`** instructions in the program code, however, because **`fence`** instructions hamper performance, programmers try to limit their usage. This however, causes a direct tension between correctness and optimisation possibilities. For example, it has been shown that to ensure linearizability of many data structures under relaxed memory, there are "laws of order" that force **`fence`** instructions to be used, and hence, linearizability itself has become a bottleneck to efficiency [3]. In the face of this result, we once again look to define suitable correctness conditions weaker than linearizability [12, 32].

Although numerous correctness conditions exist (and more are proposed each year), a unified framework within which different correctness conditions can be defined and formally compared has thus far not been developed. With the advent of correctness conditions for relaxed memory architectures, it is becoming difficult to judge the comparative strengths of different conditions. This paper presents a systematic study of correctness conditions for concurrent objects executed in multicore architectures. We do not aim to characterise the memory models themselves (for such a study see [2]), but rather characterise properties of a concurrent object executing in some memory architecture.

We make the following contributions.

1. We develop a framework that enables one to *systematically develop and reason about correctness conditions for concurrent objects*. For each property we distinguish between its *order conditions*, which define allowable orderings of concrete operations, and *commit conditions*, which provide guarantees about the operations whose effect must have taken place. This distinction is the first, to our knowledge, providing a separation of concerns when defining correctness.
2. Within this framework, we formalise well-known correctness conditions for totally ordered memory, providing insight into the relationships between them.
3. To cope with relaxed memory, we define *partial* commitment conditions where the effects

[1] Architectures with totally ordered memory are also referred to as *sequentially consistent architectures* [26]. In this paper, we use sequential consistency to refer to a property on the histories of a concurrent object as done in [4]. Sequential consistency is formalised in Definition 10.

of some completed operation calls are delayed beyond their returns due to pending writes in the buffers of the calling processes.

4. We study a specific weak memory model, TSO, and formalise known correctness conditions for it (*weak flush consistency* [12] and *weak ξ-quiescent consistency* [32]) that are weaker than linearizability, as they allow more reorderings and only ensure partial commitment.
5. Using our framework, we develop a new condition, *fence consistency*, a non-blocking compositional condition that lies between the existing conditions for TSO memory.
6. We show that the Chase-Lev work-stealing deque [7] where the `put` operation returns without fencing is not linearizable under TSO memory, but does satisfy a weaker condition, *flush consistency*, which is a sequentially consistent version of the condition defined in [12]. This condition is strictly weaker than linearizability and stronger than *ξ-quiescent consistency* (which is the sequentially consistency version of the condition in [32]).
7. We prove a hierarchy for the correctness conditions in this paper based on order and commitment properties.

2 Background

This section provides the background for the rest of the paper; we introduce the Chase-Lev work-stealing dequeue (as defined by [7]), which serves as a running example for the rest of this paper. We also informally introduce notions of correctness for concurrent objects for totally ordered memory, and the Total Store Order memory model.

2.1 Work-Stealing Deque

Work-stealing *double ended queues* (abbreviated to *deques*) are often used for load balancing in multiprocessor systems. Each worker process has a deque, which it uses to record tasks to be performed. Thus, a worker executes `put` and `take` operations that, respectively, add tasks to and remove tasks from its deque. Load balancing is achieved by allowing other, so-called "thief" processes, whose own deques are empty, to execute `steal` operations that remove elements from the deque. To avoid contention between the worker and thief processes, `put` and `take` operate at different ends of the deque from `steal` operations – a worker adds and removes tasks at the tail, whereas thieves steal tasks from the head. Because the worker and thieves operate at different ends of the deqeue, contention between the worker and thieves occurs when the deque has one element. Resolving these cases is in general difficult [13]. Fig. 1 presents a simplified version of the Chase-Lev work-stealing deque. The shared state consists of an array, `items`, of tasks (represented as integers) and variables `Head` and `Tail`, which mark the part of the array containing the elements of the deque. The other variables are local to the operations in which they occur.

2.2 Correctness Conditions

Correctness of a concurrent object is judged with respect to an abstract sequential specification [22]. The abstract specification of the deque object implementation in Fig. 1 is given in Fig. 2, consisting of a deque variable *dq*, represented as a sequence of tasks, and atomic operations `put`, `take` and `steal`; *task* is a local variable within the `take` and `steal` operations.

An object cannot execute by itself; rather it is the *clients* of an object that execute its operations. Correctness defines a relationship between *histories* of the concrete and abstract systems, which record the interactions between the client and an object via the object's external interface. Typically, a history records invocation and return events of operation

```
   void put(int task) {
P1   tl := Tail;
P2   items[tl] := task;
P3   Tail := tl + 1; }

   int steal() {
S1   while true {
S2     hd := Head;
S3     if hd ≥ Tail
S4       return emp;
S5     task := items[hd];
S6     if cas(Head,hd,hd+1)
S7       return task; } }
```

```
    int take() {
T1    tl := Tail - 1;
T2    Tail := tl;
T3    hd := Head;
T4    if hd > tl {
T5       Tail := hd;
T6       return emp; }
T7    task := items[tl];
T8    if tl > hd
T9      return task;
T10   Tail := hd + 1;
T11   if cas(Head,hd,hd+1)
T12     return task;
T13   else return emp; } }
```

Figure 1 Chase-Lev work-stealing deque.

```
void put(int task) {
  atomic {
    dq := dq ⌢⟨ task ⟩
  } }
```

```
int steal {
  atomic {
    if dq = ⟨⟩ then
      return emp
    else
      task := head(dq) ;
      dq := tail(dq) ;
      return task } }
```

```
int take {
  atomic {
    if dq = ⟨⟩ then
      return emp
    else
      task := last(dq) ;
      dq := init(dq) ;
      return task } }
```

Figure 2 Abstract work-stealing deque.

calls. Concurrent histories may consist of both overlapping and non-overlapping operation calls, inducing a partial order on events. Correctness conditions define how, if at all, this order is maintained in the corresponding abstract history. There are several well-known existing correctness conditions for totally ordered memory [22].

- *Sequential consistency* is a simple condition requiring the order of operation calls in a concrete history for a single process to be preserved. Operation calls performed by different processes may be reordered in the abstract history even if the operation calls do not overlap in the concrete history.
- *Linearizability* strengthens sequential consistency by requiring the order of non-overlapping operations to be preserved. Operation calls that overlap in the concrete history may be reordered when mapping to an abstract history.
- *Quiescent consistency* is weaker than linearizability, but is incomparable to sequential consistency. A concurrent object is said to be quiescent at some point m in its history if none of its operations are executing at m. Quiescent consistency requires the order of operation calls separated by a quiescent point to be preserved. Operation calls that are not separated by a quiescent point may be reordered, including operations performed by the same process.

It has already been shown that the Chase-Lev deque from Fig. 1 is linearizable [7]. Since linearizability implies both sequential and quiescent consistency, the Chase-Lev deque is also both sequentially and quiescently consistent.

2.3 Total Store Order (TSO) Memory

Modern multi-core architectures use local buffers to allow more efficient use of shared memory (see Fig. 3). For optimisation purposes, many architectures only provide relaxed

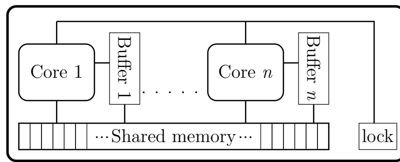


Figure 3 TSO architecture.

```
word x=0, y=0;
Process p {           Process q {
p1: x := 1 ;          q1: y := 1 ;
p2:  r1 := y }        q2: r2 := x }
```

Figure 4 TSO example.

memory guarantees. We consider Total Store Order (TSO) memory as implemented by x86 processors. A general definition of TSO is given in [1], an operational semantics in [30], and an interval-based semantics in [14].

Here, a write by a processor core is not immediately committed to shared memory. Instead it is enqueued as a pending write in the local buffer and only becomes visible to other processes after it is *flushed*, which commits the pending write in the buffer to shared memory. Hence, there is a discrepancy between the time at which a write is executed and the time at which the effect of the write becomes visible to other processes. In TSO, pending writes are flushed in a FIFO order. In addition, using a method known as *Intra-Process Forwarding* [1], when reading a memory location, a processor core fetches the value of the last pending write from its local buffer if available and from shared memory otherwise. Due to pending writes and intra-process forwarding, from an external perspective, read and write instructions within a process appear to be reordered [1], i.e., total memory order is not maintained.

▶ **Example 1.** Consider the program in Fig. 4, where processes `p` and `q` modify shared variables `x` and `y`, both of which are initialised to 0. Under totally ordered memory, when the program terminates, at least one of `r1` or `r2` would have the value 1. However, under TSO memory, it is possible for the program to terminate so that both `r1` and `r2` read the original values of `x` and `y`, i.e., both `r1` and `r2` are 0 at termination. One such execution sequence is ⟨ `p1`, `p2`, `q1`, `q2`, `flush(p)`, `flush(p)`, `flush(q)`, `flush(q)` ⟩, where `p1` denotes execution of the statement at line `p1` (similarly `p2`, etc.), and `flush(p)` denotes execution of a hardware-controlled flush event for process p (similarly `flush(q)`). The write to `x` at `p1` is not seen by process `q` until `p`'s buffer is flushed, and symmetrically for the write to `y` at `q1`. Hence, it is possible for `q` to read a value 0 for `x` at `q2` even though `q2` is executed after `p1`.

To avoid instruction reordering, a core may acquire a global *lock* (depicted in Fig. 3), which prevents all other cores from accessing shared memory. This *lock* is used to implement (coarse-grained) atomic operations such as **`cas`** [30]. In particular, a **`cas`** operation locks the buffer, performs the compare and swap, fully flushes the buffer, then releases the lock.

▶ **Example 2.** Suppose we wish to establish the postcondition that either `r1` or `r2` has value 1 for the program in Fig. 4. The only possibility is to introduce **`fence`** instructions between `p1` and `p2`, and between `q1` and `q2` in Fig. 4. Note that both **`fence`** instructions are necessary, otherwise, `r1` = `r2` = 0 remains a possible outcome of the program.

3 Defining Correctness for Concurrent Objects

The correctness conditions described in Section 2.2 are all defined in terms of an abstract sequential history which is related in a certain way to a given execution of the concurrent object in question. This relationship can be defined more precisely in terms of a mapping function that maps elements in the concrete history to those of the corresponding abstract history. Mapping functions are inspired by the encoding of linearizability in [10], which has

led to a *complete* simulation-based method for proving linearizability [29]. Our conditions for relaxed memory are also amenable to integration with such proof methods.

3.1 A Framework for Specifying Correctness

As already discussed, correctness conditions are defined in terms of histories of the abstract (sequential) and concrete (concurrent) systems. In order to make these comparable, while capturing the relevant information about concurrent executions, histories record just the invocation and return of each operation call. In a concurrent history, operation calls may be interleaved with those of other processes, so the invocation and return of a given call may be separated by any number of invocations and returns of other processes, while in a sequential history, the invocation of an operation call is immediately followed by its corresponding return.

For correctness conditions under totally ordered memory it turns out that invocation and response events are all that must be recorded. However, relaxed memory architectures often require additional events such as buffer flushes to be recorded [5, 17, 6]. Thus, assuming that process identifiers have type P, operations names have type I, and the inputs and outputs of operations have type V, we define events and histories as follows: [2]

$$Event \;\widehat{=}\; inv\langle\!\langle P \times I \times V\rangle\!\rangle \mid ret\langle\!\langle P \times I \times V\rangle\!\rangle \qquad\qquad History \;\widehat{=}\; \mathrm{seq}\, Event_C$$

where $Event_C \supseteq Event$ is the set of concrete events; the set $Event_C$ will be specialised in later sections. We say that two events e_1 and e_2 are *matching* if they form an invocation/return pair for the same operation performed by the same process:

$$matching(e_1, e_2) \;\widehat{=}\; inv?(e_1) \wedge ret?(e_2) \wedge e_1.pr = e_2.pr \wedge e_1.i = e_2.i$$

where $inv?$ and $ret?$ are true for invocation and return events, respectively, and $e.pr$ and $e.i$ denote the process and operation corresponding to an event e, respectively; similarly, $e.v$ denotes e's input/outputs. Indices m and n form a *matching pair* in a history h if they identify a pair of matching events and there is no invocation or return performed by the same process between them:

$$\begin{aligned} mp(m, n, h) \;\widehat{=}\; & matching(h(m), h(n)) \wedge \\ & \forall k : \mathrm{dom}\, h \bullet m < k < n \wedge h(k).pr = h(m).pr \Rightarrow h(k) \notin Event \end{aligned}$$

Note that in the case of totally ordered memory $Event_C = Event$, i.e., all elements of a history h are in $Event$. Hence, the second conjunct of $mp(m, n, h)$ simplifies to $\forall k : \mathrm{dom}\, h \bullet m < k < n \Rightarrow h(k).pr \neq h(m).pr$. However, this is not the case for the histories in Section 4, and there, the consequent of the second conjunct does not trivially reduce to false.

An index m is a *pending invocation* in history h if $h(m)$ is an invocation that is not followed by a matching return in h:

$$pi(m, h) \;\widehat{=}\; inv?(h(m)) \wedge \forall k : \mathrm{dom}\, h \bullet m < k \Rightarrow \neg matching(h(m), h(k))$$

A history is *sequential* if it is either empty or an alternating sequence of matching invocations and returns starting with an invocation:

$$\begin{aligned} sequential(h) \;\widehat{=}\; & h = \langle\rangle \vee (inv?(h(0)) \wedge \forall k : \mathrm{dom}\, h \bullet inv?(h(k)) \wedge k + 1 \in \mathrm{dom}\, h \Rightarrow \\ & \qquad matching(h(k), h(k + 1))) \end{aligned}$$

[2] We generally use Z mathematical notation [34]. This definition says that any element of $Event$ is of the form $inv(p, i, v)$ or $ret(p, i, v)$, where $p \in P$, $i \in I$ and $v \in V$. We also write $inv(p, op)$ for invocations with no inputs, and $ret(p, op)$ for returns with no outputs. In this paper, we assume that sequences are indexed from 0 onwards.

As in [23], we assume each process calls at most one operation at a time. We say a history h is *well formed* if each $h_{|p}$ is sequential, where $h_{|p}$ denotes a history h restricted to all events by process p; for the rest of this paper we assume that all histories are well-formed. A history h is *legal* if each return is preceded by some matching invocation:

$$legal(h) \quad \widehat{=} \quad \forall\, n : \mathrm{dom}\, h \bullet ret?(h(n)) \Rightarrow \exists\, m : \mathrm{dom}\, h \bullet m < n \wedge mp(m, n, h)$$

A correctness condition between a concurrent history h and a sequential history hs is defined in terms of a *mapping function*, $f : \mathbb{N} \nrightarrow \mathbb{N}$, which is an injective partial function from indices of h to indices of hs. Injectivity ensures that each element of h occurs at most once in hs, while partiality provides the flexibility needed to represent delayed operation calls under relaxed memory architectures (where some completed operation calls may not appear in the abstract history).

▶ **Example 3.** Consider concurrent history h and sequential history hs below:

$$\begin{aligned} h \quad &\widehat{=} \quad \langle inv(q_1, \mathtt{steal}), inv(w, \mathtt{put}, 1), ret(q_1, \mathtt{steal}, \mathtt{emp}), ret(w, \mathtt{put})\rangle \\ hs \quad &\widehat{=} \quad \langle inv(q_1, \mathtt{steal}), ret(q_1, \mathtt{steal}, \mathtt{emp})\rangle \end{aligned}$$

The mapping function from h to hs is $\{0 \mapsto 0, 2 \mapsto 1\}$. In this example, we assume that due to TSO the `put` operation has not yet taken effect.

In our framework, one only needs to define predicates on h and f; the corresponding sequential history is $hs = \{f(k) \mapsto h(k) \mid k \in \mathrm{dom}\, f\}$.

▶ **Example 4.** Sequence $h = \langle a, b, c, d\rangle$ is the set of mappings $\{0 \mapsto a, 1 \mapsto b, 2 \mapsto c, 3 \mapsto d\}$. Hence, if $f = \{2 \mapsto 0, 0 \mapsto 1, 1 \mapsto 2, 3 \mapsto 3\}$ then $hs = \{0 \mapsto c, 1 \mapsto a, 2 \mapsto b, 3 \mapsto d\} = \langle c, a, b, d\rangle$.

We distinguish between two types of predicates on h and f: *order conditions*, which describe the allowable orders of events when mapping h to hs (via f), and *commitment conditions*, which describe the events of h that must occur in hs (due to occurrence of their corresponding index in f). We write $P(\overline{v})$ if P is a predicate with free variables $\overline{v}$.

▶ **Definition 5.** Suppose $Q(h, f)$ is a predicate on history h and mapping function f, $\overline{m}$ is a vector over type $\mathbb{N}$, $P(h, \overline{m})$ is a predicate on h and $\overline{m}$, and $QR(f, \overline{m})$ and $QD(f, \overline{m})$ are predicates on f and $\overline{m}$. We say that $Q(h, f)$ is:

- an *order condition* iff $Q(h, f)$ is of the form $\forall\, \overline{m} : \mathrm{dom}\, f \bullet P(h, \overline{m}) \Rightarrow QR(f, \overline{m})$, where $QR(f, \overline{m})$ is a predicate on the range of f and $\overline{m}$ only, and
- a *commitment condition* iff $Q(h, f)$ is of the form $\forall\, \overline{m} : \mathrm{dom}\, h \bullet P(h, \overline{m}) \Rightarrow QD(f, \overline{m})$, where $QD(f, \overline{m})$ is a predicate on the domain of f and $\overline{m}$ only.

To reduce clutter, for predicates R, R_1 and R_2 on a history h, mapping function f, and boolean operator $\oplus$, we define:

$$\begin{aligned} R_1 \equiv R_2 \quad &\widehat{=} \quad \forall\, h, f \bullet legal(h) \Rightarrow R_1(h, f) = R_2(h, f) \\ R_1 \Rrightarrow R_2 \quad &\widehat{=} \quad \forall\, h, f \bullet legal(h) \wedge R_1(h, f) \Rightarrow R_2(h, f) \\ (R_1 \oplus R_1)(h, f) \quad &\widehat{=} \quad R_1(h, f) \oplus R_2(h, f) \end{aligned}$$

Using these concepts, we now define what it means for a mapping function to be valid. We formalise this definition using an order property *vmf_ord*, which ensures that for any matching pair m, n in h mapped by f, index $f(n)$ immediately follows $f(m)$:

$$vmf_ord(h, f) \quad \widehat{=} \quad \forall\, m, n : \mathrm{dom}\, f \bullet mp(m, n, h) \Rightarrow f(n) = f(m) + 1$$

and a commitment property vmf_com, which ensures that for any matching pair m, n in h, the invocation $h(m)$ is mapped by f iff the return $h(n)$ is also mapped by f:

$$vmf_com(h, f) \;\widehat{=}\; \forall\, m, n : \mathrm{dom}\, h \bullet mp(m, n, h) \Rightarrow (m \in \mathrm{dom}\, f \Leftrightarrow n \in \mathrm{dom}\, f)$$

Note that vmf_ord conforms to the structure of an order condition as defined in Definition 5, since predicate $P(h, \overline{m})$ is instantiated to $mp(m, n, h)$ and $QR(f, \overline{m})$ is instantiated to $f(n) = f(m) + 1$. Similarly, vmf_com conforms to the structure of a commitment condition as defined in Definition 5; $P(h, \overline{m})$ is instantiated to $mp(m, n, h)$ and $QD(h, \overline{m})$ instantiated to $m \in \mathrm{dom}\, f \Leftrightarrow n \in \mathrm{dom}\, f$.

We say a function f is a *valid mapping function* if, for any history h, the domain of f is contained in the domain of h, the range of f is a consecutive sequence starting from 0, only invocation/return events are mapped by f, matching pairs in h are mapped to consecutive events in the target abstract history, and f only maps matching pairs. Assuming $[m..n]$ is the set of naturals from m to n inclusive, we formalise validity for mapping functions as follows:

$$\begin{aligned} VMF(h, f) \;\widehat{=}\;& \mathrm{dom}\, f \subseteq \mathrm{dom}\, h \wedge (\exists\, n : \mathbb{N} \bullet \mathrm{ran}\, f = [0..n-1]) \wedge \\ & (\forall\, n : \mathrm{dom}\, f \bullet h(n) \in Event) \wedge vmf_ord(h, f) \wedge vmf_com(h, f) \end{aligned}$$

We can now define a correctness condition to be a conjunction of ordering and commitment conditions, along with a requirement that we have a valid mapping function.

▶ **Definition 6.** A *correctness condition* is a predicate $R(h, f)$ over a history h and mapping function f, whose definition has the form:

$$R(h, f) \;\widehat{=}\; VMF(h, f) \wedge (\textstyle\bigwedge_i OC_i(h, f)) \wedge (\bigwedge_j CC_j(h, f))$$

where each OC_i is an order condition and each CC_j is a commitment condition.

Note that the conjunct vmf_ord in VMF means that pending invocations in h are never mapped by f. However, when formalising correctness conditions, one must also consider *incomplete histories*, which contain pending invocations whose effects have already taken place and are observable to other processes [23].

▶ **Example 7.** Consider a history $HE_1 \;\widehat{=}\; \langle inv(w, \mathtt{put}, 7), inv(q, \mathtt{steal}), ret(q, \mathtt{steal}, 7)\rangle$ of the Chase-Lev deque (Fig. 1). This history is incomplete because the invocation of the `put` operation has not returned. However, its effect has clearly taken place because the `steal` operation returns 7.

To reason about such histories, Herlihy and Wing [23] consider *history extensions*, which are constructed from a history h by concatenating a sequence of returns corresponding to some of the pending invocations of h. For example, HE_1 may be extended to $HE_1 \frown \langle ret(w, \mathtt{put})\rangle$ to enable the extended history to be mapped abstractly. Note that a history may have several possible extensions. For example, for the history:

$$HE_2 \;\widehat{=}\; \langle inv(w, \mathtt{put}, 7), inv(q_1, \mathtt{steal}), ret(w, \mathtt{put}), inv(q_2, \mathtt{steal})\rangle$$

the following are some of many possible extensions:

$$\begin{aligned} HE_3 \;\widehat{=}\;& HE_2 \frown \langle ret(q_1, \mathtt{steal}, emp)\rangle \\ HE_4 \;\widehat{=}\;& HE_2 \frown \langle ret(q_2, \mathtt{steal}, 7), ret(q_1, \mathtt{steal}, emp)\rangle \end{aligned}$$

Pending invocations in an incomplete history may remain pending in the extended history. For example, in H_3, the second steal operation is still pending. Herlihy and Wing define a

function *complete* to remove all pending histories from a history, and define linearizability of a history h in terms of $complete(he)$, where he is some extension of h. However, reasoning about $complete(he)$ is often cumbersome because removal of pending invocations causes the indices of he to shift. This is exacerbated by the non-determinism of history extensions.

In our framework, because correctness is defined using an explicit mapping function, we can avoid using the *complete* function. In particular, after extending an incomplete history with return events, we can simply leave out pending invocations when mapping this extended history, simplifying the definitions and the proofs. We now lift correctness to the level of concurrent objects. This definition is tied to the fact that every concurrent object is inherently an implementation of some sequential abstract counterpart.

▶ **Definition 8.** A concurrent object C implementing an abstract object A is *correct* with respect to a correctness condition R, denoted $C \models_A R$, iff for any legal history h of C, there exists an extension he of h, a mapping function f such that $R(he, f)$ holds, and a valid sequential history hs of A such that $hs = \{f(k) \mapsto he(k) \mid k \in \operatorname{dom} f\}$.

The next theorem states that if a concurrent object implements an abstract object for some notion of correctness, then it also implements the abstract object with respect to a weaker correctness condition.

▶ **Theorem 9.** *Suppose C is a concurrent object, A an abstract object and R_1, R_2 are correctness conditions such that $R_1 \Rrightarrow R_2$. If $C \models_A R_1$ then $C \models_A R_2$.*

The proof is straightforward by expanding the definitions and using the fact that *legal* is *extension closed*, i.e., if $legal(h)$ holds and he is an extension of h, then $legal(he)$ holds.

3.2 Specifying Correctness Conditions for Totally Ordered Memory

We now use our framework to formalise the conditions for totally ordered memory from Section 2.2: sequential consistency, linearizability and quiescent consistency. There are already existing formalisations of each of these in the literature, e.g., using partial orders. However, using our framework, we are able to distinguish between the different types of properties that form each condition.

Each correctness condition in Section 2.2 implies a *total* commitment condition, which means that all completed operation calls in a given history h must be mapped by f to some operation call in a sequential history.

$$total(h, f) \quad \widehat{=} \quad \forall\, m : \operatorname{dom} h \bullet h(m) \in Event \wedge \neg pi(m, h) \Rightarrow m \in \operatorname{dom} f$$

Sequential consistency is defined in terms of an order condition sc, which states operation calls in h by the same process are not reordered by f when mapped to a sequential history.

$$\begin{aligned} sc(h, f) \quad \widehat{=} \quad \forall\, m, n : \operatorname{dom} f \bullet\ & m < n \wedge h(m).pr = h(n).pr \wedge \\ & ret?(h(m)) \wedge inv?(h(n)) \Rightarrow f(m) < f(n) \end{aligned}$$

▶ **Definition 10.** A concurrent object C implementing an abstract object A is *sequentially consistent* iff $C \models_A SC$, where [3] $SC \mathrel{\widehat{=}} VMF \wedge sc \wedge total$.

[3] Note that by definition of $\equiv$ and pointwise lifting, $SC(h, f) \equiv VMF(h, f) \wedge sc(h, f) \wedge total(h, f)$ for any history h and mapping function f.

Linearizability [23] is a straightforward extension to sequential consistency, strengthening the order condition so that an operation call is not reordered with another operation call that is invoked after the first operation returns.

$$lin(h, f) \quad \widehat{=} \quad \forall\, m, n : \mathrm{dom}\, f \bullet m < n \wedge ret?(h(m)) \wedge inv?(h(n)) \Rightarrow f(m) < f(n)$$

▶ **Definition 11.** A concurrent object C implementing an abstract object A is *linearizable* iff $C \models_A LIN$, where $LIN \mathrel{\widehat{=}} VMF \wedge lin \wedge total$.

It is straightforward to link this definition to the formalisation by Derrick *et al* [10], which has in turn been linked with Herlihy and Wing's original definition.

Quiescent consistency, as informally described by Shavit [31], has been formalised in [9] and is defined in terms of bijections between a concurrent history and its corresponding abstract history. We first define a *quiescent point* as an index m in a history h at which there are no pending invoked operation calls. We use $h[m..n]$ to denote the *projection* of the elements of h from index m to n, inclusive, i.e., $h[m..n] = \langle h(m), h(m+1), \dots, h(n-1), h(n)\rangle$.

$$qp(m, h) \quad \widehat{=} \quad \forall\, n : \mathrm{dom}\, h \bullet (n \leq m => \neg pi(n, h[0..m]))$$

The ordering condition for quiescent consistency states that f does not reorder two indices in h separated by a quiescent point.

$$qc_ord(h, f) \quad \widehat{=} \quad \forall\, m, k, n : \mathrm{dom}\, f \bullet m < k < n \wedge qp(k, h) \Rightarrow f(m) < f(n)$$

▶ **Definition 12.** A concurrent object C implementing an abstract object A is *quiescent consistent* iff $C \models_A QC$, where $QC \mathrel{\widehat{=}} VMF \wedge qc_ord \wedge total$.

A benefit of our formalisation is that it is now straightforward to formally prove that linearizability implies both sequential consistency and quiescent consistency, the former is because $lin \Rrightarrow sc$ holds, while the latter is because $lin \Rrightarrow qc$. It is well known that $SC \Rrightarrow LIN$ and $QC \Rrightarrow LIN$ are both false; constructing counter-examples is straightforward [22].

4 Correctness Conditions for Total Store Order Memory

We now explore notions of correctness for concurrent objects in relaxed memory architectures. In particular, we focus on the potential for optimisation for TSO architectures. To simplify development of correctness conditions, we present each correctness condition as an instantiation of a number of high-level steps. We formalise two recently defined notions of correctness [12, 32], develop sequentially consistent variations of these, then develop a new correctness condition, *fence consistency*.

4.1 Minimising `fence` instructions in TSO

Correctness conditions that hold for a concurrent object under totally ordered memory may no longer hold in the presence of relaxed memory. For our running example, under TSO memory, consider the following scenario. After initialisation, suppose two complete `put` operations as well as their `flush`es have been executed. Thus, the deque is of size two with tasks a_0 and a_1 at array indices 0 and 1, and `Head` $= 0$ and `Tail` $= 2$. Suppose w invokes a `take`, which executes up to line `T4` without executing any `flush`es, setting its local variables `hd` and `tl` to 0 and 1, respectively. Now suppose two thief processes q_1 and q_2 invoke and execute `steal` operations up to completion, stealing both a_0 and a_1. The worker may now continue executing `take`, and return some unspecified value for `task` because the test at `T8`

```
    int take() {                      T8   task := items[tl];
T1  tl := Tail - 1;                   T9   if tl > hd
T2  Tail := tl;                       T10    return task;
T3  fence ;                           T11  Tail := hd + 1;
T4  hd := Head;                       T12  if cas(Head,hd,hd+1)
T5  if hd > tl {                      T13    return task;
T6     Tail := hd;                    T14  else return emp; } }
T7     return emp; }
```

Figure 5 Chase-Lev `take` operation modified for TSO.

succeeds. Such an execution cannot be proved to implement Fig. 2 for any sensible definition of correctness.

Liu et al. [27] have shown that linearizability can be restored provided (i) a **fence** is introduced immediately after `P3` in the `put` operation, and (ii) the `take` operation in Fig. 1 is replaced by the `take` in Fig. 5, where a **fence** has been introduced after `T2`. As memory barriers in the form of **fence** instructions are expensive, our question is: *Are there conditions weaker than linearizability that would allow only one **fence** to be used such that the behaviours one obtains are still sensible?* Although removing a single **fence** instruction may not seem like a big change, because a client may execute several **put** operations consecutively, there is a potential for a high level of efficiency gains. Furthermore, since data structures such as deques are used to implement underlying system mechanisms such as schedulers [16] and operating system kernels [28], avoiding **fence** instructions can provide system-wide benefits.

It turns out that a **fence** after `T2` is needed to avoid the scenario described above. In the other case, it turns out that the object is not linearizable, because the following is possible:

$$\langle inv(w, \texttt{put}, x), ret(w, \texttt{put}), inv(q, \texttt{steal}), ret(q, \texttt{steal}, emp)\rangle \qquad (1)$$

This occurs because the effect of a `put` operation only occurs after the write at `P3` is flushed. Therefore, the `steal` operation may read an older value causing it to return `emp`. We argue that such histories should be allowed – it is perfectly sensible for the `steal` and `put` operations to be reordered because the effect of the `put` has merely been delayed by buffer effects, whereby the `put` operation continues to execute beyond its return event. We therefore, set out to formally define correctness conditions that would accept histories such as (1), e.g., for the Chase-Lev deque under TSO memory where no **fence** instructions are introduced after `P3`.

Behaviours in TSO memory in which the effect of an operation is delayed beyond its return are already accepted as being correct for many implementations, e.g., spinlock [12, 28], Burns' mutex [35] and the sequence lock [32]. However, a precise notion of correctness in these scenarios has thus far not been developed. Even less is known about the implications of accepting more histories than allowed by linearizability.

4.2 Defining Correctness Conditions

It turns out that there are several possibilities for interpreting correctness for delayed operations. We describe a sequence of steps for defining correctness conditions for TSO memory, where each step identifies an aspect of the condition that must be considered. Picking a particular instantiation at each step, leads to a particular correctness condition.

▶ Step 1 (Determine the events to be recorded in histories). For the conditions on totally ordered memory in Section 2.2, histories only needed to record invocation/response events.

For TSO memory, it is often necessary to record additional events, with rules on how these events are recorded. Formally, these additional events are recorded by instantiating $Event_C$.

For example, for weak ξ-quiescent consistency (as defined in Section 4.3), we record an additional event $\xi(p)$, thus $Event_C ::= Event \mid \xi\langle\langle P \rangle\rangle$. Event $\xi(p)$ is triggered (i.e., recorded) if either (i) a transition causes the buffer of process p to become empty, or (ii) if process p returns from an operation call when p's buffer is empty. For case (i), $\langle \xi(p) \rangle$ is concatenated, while for case (ii) the two-event sequence $\langle ret(p, op, v), \xi(p) \rangle$ is concatenated to the end of the history[4]. Note that a transition causing p's buffer to become empty may be caused by a CPU-controlled buffer flush, which may occur after p has already returned. We assume $\xi(p)$ is not recorded if the buffer of p is already empty in the prestate of a non-return transition, and if p returns when its buffer is non-empty, then only $\langle ret(p, op, v) \rangle$ is concatenated to the end of the history. □

We explain the next two steps assuming $Event_C ::= Event \mid \xi\langle\langle P \rangle\rangle$ has been fixed as defined in Step 1. For our examples below, we assume that the deque is initially empty, w denotes the worker process, and q, q_1, q_2 and q_3 denote thief processes.

▶ Step 2 (Determine what operations can be reordered). A common feature of the correctness conditions discussed in Section 2.2 is that operation calls whose active intervals overlap may be reordered. For totally ordered memory, an operation call may be considered to be active from its invocation to its return.

In the context of TSO memory, because some operation calls may return with non-empty buffers, there is additional flexibility in defining what counts as an active operation [12, 32, 35]. One possibility is to think of an operation call by a process p as being active until buffer of process p becomes empty. Consider the following history, which is possible for the deque in Fig. 1 but with the `take` operation from Fig. 5.

$$\begin{aligned} HC_1 \;\widehat{=}\; & \langle inv(w, \texttt{put}, x), ret(w, \texttt{put}), inv(q_1, \texttt{steal}), ret(q_1, \texttt{steal}, emp), \xi(q_1), \\ & inv(q_2, \texttt{steal}), ret(q_2, \texttt{steal}, emp), \xi(q_2), \xi(w), inv(q_3, \texttt{steal}), \\ & \xi(q_3), ret(q_3, \texttt{steal}, x), \xi(q_3)\rangle \end{aligned}$$

HC_1 cannot be linearized with respect to the abstract deque (Fig. 2) – HC_1 restricted to invocations and responses only is sequential and the `steal` occurs after the `put` has completed, yet the `steal` returns empty. However, in the context of TSO memory with the interpretation that returned operations calls by process p are active until p's buffer is empty, HC_1 can be explained by the following sequential history:

$$\begin{aligned} & \langle inv(q_1, \texttt{steal}), ret(q_1, \texttt{steal}, emp), inv(q_2, \texttt{steal}), ret(q_2, \texttt{steal}, emp), \\ & inv(w, \texttt{put}, x), ret(w, \texttt{put}), inv(q_3, \texttt{steal}), ret(q_3, \texttt{steal}, x)\rangle \end{aligned}$$ □

It turns out that there are varying ways of defining active operations. In this paper we explore two possibilities: the first (inspired by [32]) allows an operation call to be active as long as the buffer of its calling process is non-empty, and the second (inspired by [12]) is more restricted, allowing an operation call to be active only as long as the final write corresponding to the operation call has not been flushed.

▶ Step 3 (Determine the commitment conditions). The conditions in Section 2.2 for totally ordered memory are all total, i.e., any operation call that has returned must be mapped to some abstract operation call. Total conditions are appropriate for such architectures because

[4] Note that there are other alternatives to recording case (ii) in the history; e.g., one could use a special "return empty" event that is distinct from ret events to obtain $Event_C ::= Event \mid ret_\xi\langle\langle P \times O \times V \rangle\rangle \mid \xi\langle\langle P \rangle\rangle$.

hardware guarantees that each write is immediately committed to shared memory when the write instruction is executed, making its effect visible to other concurrent threads.

On the other hand, in relaxed memory models, write instructions may be cached in local buffers, and thus not seen by other processes until the buffers are flushed. Hence, when an operation call returns, the effect of the operation may not have have occurred in shared memory. We refer to a returned operation call that has taken effect as a *committed* operation call and as *uncommitted*, otherwise. To take delayed operation calls (due to buffer effects) into account, we allow correctness conditions to be defined using *partial commitment conditions*, allowing some completed operation calls to not be mapped to any abstract operations. When specifying partial commitment conditions, it turns out that one must additionally define conditions that dictate when an operation must become committed.

For TSO memory, one possible instantiation of this step is to require that *all operation calls of process p that have returned prior to $\xi(p)$ occurring must have committed.* For example, consider the following history:

$$\begin{aligned} HC_2 \;\widehat{=}\;& \langle inv(w, \mathtt{put}, x), ret(w, \mathtt{put}), \xi(w), inv(w, \mathtt{put}, y), ret(w, \mathtt{put}), \\ & inv(q, \mathtt{steal}), ret(q, \mathtt{steal}, x), \xi(q)\rangle \end{aligned}$$

History HC_2 cannot be judged consistent against sequential histories $\langle\,\rangle$ or $\langle inv(w, \mathtt{put}, x), ret(w, \mathtt{put})\rangle$ because due to $\xi(w)$, the first `put` operation must be committed, and due to $\xi(q)$, the `steal` must have also been committed. Note that $\xi(p)$ represents that latest point at which commitments of completed operations of process p must occur; the commitment condition does not prevent operations from committing earlier. Thus, for example, both sequential histories below satisfy the requirement:

$\langle inv(w, \mathtt{put}, x), ret(w, \mathtt{put}), inv(q, \mathtt{steal}), ret(q, \mathtt{steal}, x)\rangle$

$\langle inv(w, \mathtt{put}, x), ret(w, \mathtt{put}), inv(q, \mathtt{steal}), ret(q, \mathtt{steal}, x), inv(w, \mathtt{put}, y), ret(w, \mathtt{put})\rangle$ ☐

Note that the conditions in Section 2.2 can also be defined using these three steps. For all three conditions, $Event_C ::= Event$ (Step 1), and the commitment condition is *total*, which states completed operation calls must appear in any corresponding sequential history (Step 3). The three conditions only differ in terms of their order properties, *sc*, *lin* and *qc*, which are different instantiations of Step 2.

4.3 Weak ξ-Quiescent Consistency

Smith et al. [32] prove correctness of a sequence lock algorithm in TSO memory with respect to quiescent consistency [31]. An object is considered to be quiescent in a history if none of its operations calls are pending in the history and the buffer of each process that has called an operation of the object is empty. Note that the buffer of a process calling an operation may become empty after the operation has returned. Reordering of operations across a quiescent point is disallowed.

This condition may be formalised by instantiating the steps in Section 4.2. For Step 1, we use $Event_C ::= Event \mid \xi\langle\langle P\rangle\rangle$ because we must reason about empty buffers. We say such a history h is *legal* iff $h_{|Event}$ (i.e., h restricted to invocations and return events) is legal. Histories of this type are extension closed. For Step 2, as in [32], we say an operation call is active until the object becomes quiescent. Finally, for Step 3, we require that all operation calls be committed when the object becomes quiescent – until then operation calls remain uncommitted.

An index m is quiescent iff the last completed operation call for each process p has been followed by $\xi(p)$. We say that $p \in P$ is *quiescent between* indices m and n of history h iff m is

a return for p, the buffer of p becomes empty at some point between m and n, and p does not invoke any new operation between m and n. Thus, we define the following, where $empty?(e)$ holds iff $e = \xi(p)$ for some $p \in P$, and $inv_p?(e) \mathrel{\widehat=} inv?(e) \wedge e.pr = p$, and predicates $ret_p?$ and $empty_p?$ are similarly defined.

$$qb(m, n, p, h) \mathrel{\widehat=} m < n \wedge ret_p?(h(m)) \wedge (\exists k : m + 1..n \bullet h(k) = \xi(p)) \wedge (\forall k : m + 1..n \bullet \neg inv_p?(h(k)))$$

Using qb, we define a quiescent point m in history h as follows.

$$qp_\xi(m, h) \mathrel{\widehat=} \forall p : P, n : \mathrm{dom}\, h \bullet n < m \wedge inv_p?(h(n)) \Rightarrow \exists k : \mathrm{dom}\, h \bullet qb(k, m, p, h)$$

Then we define the order and commitment conditions for weak ξ-consistency as follows. Order condition wqc_ord_ξ states that two events separated by a quiescent point are not reordered.

$$wqc_ord_\xi(h, f) \mathrel{\widehat=} \forall m, n : \mathrm{dom}\, f \bullet (\exists k : \mathbb{N} \bullet m < k < n \wedge qp_\xi(k, h)) \Rightarrow f(m) < f(n)$$

The commitment condition wqc_com_ξ requires all operation calls occurring before a quiescent point to be committed.

$$wqc_com_\xi(h, f) \mathrel{\widehat=} \forall m, n : \mathrm{dom}\, h \bullet m \leq n \wedge qp_\xi(n, h) \wedge h(m) \in Event \Rightarrow m \in \mathrm{dom}\, f$$

▶ **Definition 13.** A concurrent object C implementing an abstract object A is *weakly ξ-quiescent consistent* iff $C \models_A WQC_\xi$, where $WQC_\xi \mathrel{\widehat=} VMF \wedge wqc_ord_\xi \wedge wqc_com_\xi$.

Weak ξ-quiescent consistency is a straightforward generalisation of quiescent consistency, and hence, we omit example histories here. This definition is weak because it does not guarantee sequential consistency, i.e., operations calls within a process may be reordered. However, weak ξ-quiescent consistency follows the condition defined in [32], which in turn is based on quiescent consistency [9, 22, 31]. In Section 4.5, we motivate a new correctness condition, then strengthen WQC_ξ accordingly so that it guarantees sequential consistency. Defining such variations in our framework is straightforward.

4.4 Weak Flush Consistency

We now formalise the correctness condition defined by Derrick et al. [12], which we refer to in this paper as *weak flush consistency*. Informally, weak flush consistency captures the idea that an operation call may be considered to be active only until the last pending write corresponding to the operation call is flushed. This differs from weak ξ-quiescent consistency, since an operation call may become inactive even if the buffer of the calling process is non-empty.

Derrick et al. [12] define weak flush consistency in terms of linearizability of *transformed* histories. Their (deterministic) transformation algorithm proceeds as follows: (i) the final flush corresponding to each operation call is located, (ii) the actual return is moved to this flush, and (iii) all remaining flushes are removed from the history. The standard definition of linearizability is then applied to the transformed histories. For example, consider the following history, where $\phi^k(p)$ denotes k consecutive flush events of process p:

$$\langle inv(w, \texttt{put}, x), ret(w, \texttt{put}), inv(w, \texttt{put}, y), inv(q, \texttt{steal}), \phi(q), \phi^3(w), ret(q, \texttt{steal}, emp), ret(w, \texttt{put})\rangle$$

This history is transformed to

$$\langle inv(w, \texttt{put}, x), inv(w, \texttt{put}, y), inv(q, \texttt{steal}), ret(w, \texttt{put}), ret(q, \texttt{steal}, emp), ret(w, \texttt{put})\rangle$$

then judged consistent because it is linearizable with respect to the sequential history:

$$\langle inv(q, \mathtt{steal}), ret(q, \mathtt{steal}, emp), inv(w, \mathtt{put}, x), ret(w, \mathtt{put}), inv(w, \mathtt{put}, y), ret(w, \mathtt{put})\rangle$$

Using our framework, we can define weak flush consistency directly, i.e., without performing such a history transformation. This involves two small extensions to the *Event* type. First, we record each flush event. Second, because the number of writes each operation call performs is potentially non-deterministic, we additionally record each write event to identify the last flush corresponding to each operation call. Thus, for Step 1, we define $Event_C ::= Event \mid \omega\langle\langle P\rangle\rangle \mid \phi\langle\langle P\rangle\rangle$. Here, $\omega(p)$ denotes a write by process p, and $\phi(p)$ records a flush for process p. We assume $write?(e)$ and $flush?(e)$ hold iff event e is a write and flush, respectively. We assume that only writes and flushes executed by the concurrent object in question are recorded in the histories, and that writes are executed between matching invocations and responses. Hence, we say such a history h is *legal* iff $h_{|Event}$ is legal and $\omega(p)$ only occurs in h when p is executing some operation. Legality of histories of this type are also extension closed.

For Step 2, we say an operation call that completes after executing l writes remains active until each of these l writes have been flushed, or until the return occurs, whichever is later. To formalise this, we define a function *num*, which counts the number of events in h that satisfy event predicate *ep* (mapping an event to a boolean) up to and including index m:

$$num(ep, m, h) \quad \widehat{=} \quad size(\{h(k) \mid 0 \leq k \leq m \wedge ep(h(k))\})$$

The order condition for weak flush consistency counts the number of writes, say l, that have occurred when an operation call, say L, by process p returns. Any operation invoked after these l flushes have occurred may not be reordered with L. Thus, we obtain:

$$\begin{aligned} wflc_ord(h, f) \quad \widehat{=} \quad & \forall\, m, n : \mathrm{dom}\, f \bullet \\ & \begin{pmatrix} \exists\, p : P \bullet m < n \wedge ret_p?(h(m)) \wedge inv?(h(n)) \wedge \\ \qquad num(write_p?, m, h) \leq num(flush_p?, n, h) \end{pmatrix} \Rightarrow f(m) < f(n) \end{aligned}$$

Note that $num(write_p?, m, h)$ counts all writes by process p since initialisation.

For Step 3, we say that completed operation calls whose last write has been flushed must commit. Weak flush consistency has two commitment conditions. The first condition, $wflc_com_1$, states that an operation call is committed whenever all writes executed by that operation call have been flushed and the call returns. The second, $wflc_com_2$, requires than an operation call L that executes l writes that are not flushed before L returns is committed whenever l flushes of the calling process have occurred.

$$\begin{aligned} wflc_com_1(h, f) \quad \widehat{=} \quad & \forall\, n : \mathrm{dom}\, h \bullet \\ & \begin{pmatrix} \exists\, p : P \bullet ret_p?(h(n)) \wedge \\ \qquad num(flush_p?, n, h) = num(write_p?, n, h) \end{pmatrix} \Rightarrow n \in \mathrm{dom}\, f \\ wflc_com_2(h, f) \quad \widehat{=} \quad & \forall\, k, n : \mathrm{dom}\, h \bullet \\ & \begin{pmatrix} \exists\, p : P \bullet n < k \wedge ret_p?(h(n)) \wedge flush_p?(k) \\ \qquad num(write_p?, n, h) = num(flush_p?, k, h) \end{pmatrix} \Rightarrow n \in \mathrm{dom}\, f \end{aligned}$$

▶ **Definition 14.** A concurrent object C implementing an abstract object A is *weakly flush consistent* iff $C \models_A WFLC$, where $WFLC \mathrel{\widehat{=}} VMF \wedge wflc_ord \wedge wflc_com_1 \wedge wflc_com_2$.

▶ **Example 15.** Consider the histories below, neither of which is linearizable, where $\omega^k(p)$ denotes k consecutive $\omega(p)$ events.

$$\langle inv(w, \texttt{put}, x), \omega^3(w), ret(w, \texttt{put}), \phi^2(w), inv(q, \texttt{steal}), \phi(w), \omega^3(q), \phi^3(q), ret(q, \texttt{steal}, emp)\rangle \quad (2)$$

$$\langle inv(w, \texttt{put}, x), \omega^3(w), ret(w, \texttt{put}), \phi^3(w), inv(q, \texttt{steal}), \omega^3(q), \phi^3(q), ret(q, \texttt{steal}, emp)\rangle \quad (3)$$

History (2) is weakly flush consistent; the `steal` operation is invoked before the final flush of the `put` occurs, and hence the active intervals of the `put` and `steal` overlap, allowing the `steal` to be ordered before the `put`, i.e., (2) is flush consistent with respect to sequential history $\langle inv(q, \texttt{steal}), ret(q, \texttt{steal}, emp), inv(w, \texttt{put}, x), ret(w, \texttt{put})\rangle$. On the other hand, (3) is not weakly flush consistent because the active intervals of `put` and `steal` do not overlap, namely, the `steal` is invoked after the final flush of `put` has occurred, and hence, cannot be ordered before the `put`.

4.5 Sequential Consistency for TSO Memory

Both weak ξ-quiescent consistency and weak flush consistency allow operation calls for the same process to be reordered, i.e., sequential consistency may be violated. If sequential consistency is required, the following history should be judged incorrect even though both `put` operation calls are "active" over the interval in which the `steal` occurs.

$$HC_3 \;\widehat{=}\; \langle inv(w, \texttt{put}, x), ret(w, \texttt{put}), inv(w, \texttt{put}, y), ret(w, \texttt{put}), inv(q, \texttt{steal}), \xi(q), ret(q, \texttt{steal}, y), \xi(q), \xi(w)\rangle$$

Because `put` operations add elements to the end of the deque and `steal` operations remove elements from the beginning, the only way to explain HC_3 is by reordering the first two `put` calls, violating sequential consistency. It turns out that sequential consistency is an important property. In fact, it is equivalent to *observational refinement* between a concrete object and its abstract specification for *data independent* clients [15]. The notion of observational refinement is based on observing the initial and final values of variables of client programs.

> An implementation C of a data structure is an *observational refinement* of an implementation A of the same data structure, if every observable behaviour of any client program using C can also be observed when the program uses A instead. [15, pg 412]

Data independence states that each process accesses only local variables or resources in its client operations. We therefore define sequentially consistent versions of weak ξ-quiescent consistency and weak flush consistency. This is straightforward in our framework and involves adding the already defined order condition sc as a conjunct, i.e., the sequentially consistent versions are obtained via a different instantiation of Step 2.

▶ **Definition 16.** Suppose C is a concurrent object implementing an abstract object A. We say C is

- ξ*-quiescent consistent* iff $C \models_A QC_\xi$, where $QC_\xi \;\widehat{=}\; WQC_\xi \wedge sc$
- *flush consistent* iff $C \models_A FLC$, where $FLC \;\widehat{=}\; WFLC \wedge sc$.

It is possible to prove that the deque in Fig. 1 is flush consistent under TSO memory. The argument is complex and combines the most sophisticated types of reasoning from linearizability proofs (i.e., those that require reasoning about future behaviour [29, 21] and about linearization points in other operations [11, 8]) with the additional complexities of reasoning about delayed operations [12, 32]. We use the term *commit point* to refer to the atomic program statement that causes the effect of an operation to be felt abstractly; this is analogous to a *linearization point* in a linearizability proof [10, 8]. We provide a proof sketch for this argument below.

▶ **Proposition 17.** *The work-stealing deque in Fig. 1 is flush consistent under TSO with respect to the abstract deque in Fig. 2.*

Proof. A standard representation relation is used to relate the concrete array used by Fig. 1 with the abstract sequence in Fig. 2. We assume the existence of a program counter variable pc_p for each process p, with value *idle* if p is currently not invoking any operation and value $P1, \ldots, P3, S1, \ldots S7, T1, \ldots T14$ corresponding to the labels in Fig. 1, otherwise.

To cope with delayed operations, we record operation calls that have returned without committing in an auxiliary variable $g \in P \to \text{seq}\, Event$. Each invocation by process p appends a corresponding invoke event in $g(p)$, and each return that has not committed appends the corresponding return event to $g(p)$. Therefore, $g(p)$ records the sequence of uncommitted calls by p in real-time order. To ensure sequential consistency, pending operations are committed by process p by removing the first two elements from $g(p)$ (which must be an invocation/return), then executing the corresponding operation abstractly. The commit points of the algorithm are as follows.

- A `flush` by a worker process of a pending write to *Tail* when $pc_w \in \{idle, T1, T2\}$. The only possible pending operation in this case is `put`, which is committed.
- A `flush` by a worker process of a pending write to *Tail* when $pc_w = T3$, which commits the first pending operation $g(w)$. There are two cases depending on the value of $head(g(w))$. If $head(g(w)).i =$ `put`, then the `flush` corresponds to committing a completed `put` (that has already returned). Otherwise, there are 3 further cases. The two simpler cases are: (i) if $Tail > Head$ in the post state, then the `flush` must commit the currently executing `take` operation, and (ii) if $Tail < Head$, then the `flush` must commit a `take` that returns *emp*. The difficult case is if $Tail = Head$ in the post state, which signifies the case where there is only one task in the deque, causing the current `take` operation to race with a `steal` operation executed by another process. If there is an active `steal` operation at `S4` or `S5`, there are two possible outcomes, depending on the *future* execution of the program:
  - the active `steal` operation succeeds and the `take` returns *emp* (via `T14`), or
  - the `take` succeeds and the `steal` tries again.

  One of the two outcomes must be determined at this `flush` because any other operations `steal` invoked (by a third process) after the `flush` has been executed will return *emp*. Note that if there is no active `steal` operation at line `S4` or `S5`, then the only possible future outcome is that the `take` succeeds.
- The `fence` at `T3`. This commits all pending `put` operations. Then, commits the executing `take`, or a `take` and a `steal` depending on whether $Tail > Head$, $Tail < Head$ or $Tail = Head$ holds in the post state, as in the `flush` case described above.
- A successful `cas` at `S6`. Assuming the `steal` is executed by a process $p \neq w$, there are two cases, depending on the value of $g(p)$. If $g(p) = \langle inv(p, steal, \perp)\rangle$, then the `steal` has not yet been committed earlier, and the successful `cas` commits the `steal`. Otherwise, $g(p) = \langle ret(p, steal, tout)\rangle$ holds, i.e., the `steal` has been committed earlier (by a `flush` or `fence` as above), and hence, the `cas` is not a commit point.

◀

Interestingly, the `cas` at `T12` is not a commit point because the `take` operation must have already been committed either at the `fence` at `T3` or an earlier `flush`.

Using a weaker condition than linearizability has allowed us to prove correctness with respect to a standard sequential abstract specification. This differs from [6, 17], where linearizability is established, but the abstract specification differs from what one would

expect. In particular, [6] uses an abstraction that executes using TSO semantics, whereas [17] includes additional non-determinism to cope with buffer effects at the concrete level.

4.6 Fence Consistency

ξ-quiescent consistency is simple, but provides relatively weak guarantees about a program's behaviour; flush consistency on the other hand is a conservative weakening of linearizability (providing strong behavioural guarantees), but is relatively complex as both writes and flushes need to be recorded in a history. Following the formalisations in the preceding sections, we identify a new correctness condition, *fence consistency*, that is weaker than flush consistency, but stronger than ξ-quiescent consistency. The condition aims to capture the fact that in many cases, if the buffer of a process p becomes empty at say index m of a history, then completed operation calls for p before m should not be reordered with operations invoked after m. This means that the event corresponding to a "buffer becoming empty" is a barrier to reordering, which is precisely the intention of a programmer specified **fence** instruction. This condition is potentially useful for developing algorithms that offer more optimisation possibilities than flush consistency.

Fence consistency has the characteristics of both flush consistency and ξ-quiescent consistency. Like ξ-quiescent consistency, for Step 1 we instantiate $Event_C ::= Event \mid \xi\langle\langle P\rangle\rangle$ and record $\xi(p)$ for $p \in P$ events in the same way. Step 2 is similar to flush consistency: we say an operation call is active until the buffer of the calling process is empty; operation calls may only be reordered if the intervals in which they are active overlap. For Step 3, we require operation calls executed by process p to be committed when p's buffer becomes empty. Finally, we require sequential consistency. The ordering condition for fence consistency states that if $\xi(p)$ occurs at an index k of history h, then any operation calls of process p completed (i.e., returned) before k are not reordered with an operation call by any process invoked after k, i.e.,

$$\begin{aligned} fc_ord(h,f) \;\widehat{=}\;\; & \forall\, m,n : \mathrm{dom}\, f \bullet ret?(h(m)) \wedge inv?(h(n)) \wedge \\ & (\exists\, k : \mathrm{dom}\, h \bullet m < k < n \wedge empty?(h(k)) \wedge h(m).pr = h(k).pr) \\ & \Rightarrow f(m) < f(n) \end{aligned}$$

Furthermore, if $\xi(p)$ occurs at an index k of history h, then all completed operation calls of p occurring before k must have been committed, i.e.,

$$\begin{aligned} fc_com(h,f) \;\widehat{=}\;\; & \forall\, n,k : \mathrm{dom}\, h \bullet n < k \wedge ret?(h(n)) \wedge empty?(h(k)) \wedge \\ & h(n).pr = h(k).pr \Rightarrow n \in \mathrm{dom}\, f \end{aligned}$$

▶ **Definition 18.** A concurrent object C implementing an abstract object A is *fence consistent* iff $C \models_A FC$, where $FC \mathrel{\widehat{=}} VMF \wedge sc \wedge fc_ord \wedge fc_com$.

▶ **Example 19.** In a fence consistent history, $\xi(p)$ does not prevent an operation call for p that is still executing from being reordered. This is for good reason. For example, consider the following history of the Chase-Lev deque:

$$\langle inv(w,\texttt{put},x), \xi(w), ret(w,\texttt{put}), inv(q,\texttt{steal}), ret(q,\texttt{steal},emp), \xi(q), \xi(w)\rangle \qquad (4)$$

Here, we know that even though an $\xi(w)$ occurs between $inv(w,\texttt{put},x)$ and $ret(w,\texttt{put})$, process w's buffer is non-empty when the `put` returns because $ret(w,\texttt{put})$ is not immediately followed by $\xi(w)$. Therefore, a `steal` operation may read an old value of *Tail*. Fence consistency states that the `put` operation is active until the second $\xi(w)$ occurs, allowing (4) to be judged consistent with respect to to the sequential history:

$$\langle inv(q,\texttt{steal}), ret(q,\texttt{steal},emp), inv(w,\texttt{put},x), ret(w,\texttt{put})\rangle \qquad \square$$

▶ **Example 20.** Fence consistency does not require an operation call to commit unless $\xi(p)$ occurs after the operation call has returned. For example, the history

$$\langle inv(w, \texttt{put}, x), \xi(w), ret(w, \texttt{put}), inv(q, \texttt{steal}), ret(q, \texttt{steal}, emp), \xi(q)\rangle \tag{5}$$

is fence consistent with respect to history $\langle inv(q, \texttt{steal}), ret(q, \texttt{steal}, emp)\rangle$, where the effect of the put has not yet been reflected abstractly.

The next two theorems establish that fence consistency is both *non-blocking* and *compositional*. The non-blocking property pertains to *total operations*, which are operations for which a return is well-defined in any system state. A correctness condition is non-blocking iff total operations can always complete, i.e., are never prevented from completing by the correctness condition itself. A correctness condition R is compositional if for any multi-object system, the system as a whole satisfies R iff each object of the system satisfies R, which ensures that R can be proved in a modular manner. The lack of compositionality of sequential consistency was a key motivation for Herlihy and Wing to introduce linearizability, which is compositional [23], and hence, we see compositionality as being an important property.

▶ **Lemma 21.** *Suppose $m \in \mathit{dom}\, h$ such that $pi(m, h)$ holds, $h(m).i$ is a total operation, and e is an event such that $matching(h(m), e)$ holds. Then, $FC(h, f) \Rightarrow \exists f' \bullet FC(h \frown \langle e\rangle, f')$.*

▶ **Lemma 22.** *Suppose h is a history, f is a mapping function, hr is a sequence of returns and hr' a permutation of hr. Then $FC(h \frown hr, f) \Rightarrow \exists f' \bullet FC(h \frown hr', f')$.*

▶ **Theorem 23** (Fence consistency is non-blocking). *Suppose $FC(he, f)$, where he extends history h and f is a matching function. If $m \in \mathit{dom}\, h$ is an index such that $pi(m, h)$ and $h(m).i$ is a total operation, then there exists an event e such that $matching(h(m), e)$, an extension he' of $h \frown \langle e\rangle$, and a mapping function f' such that $FC(he', f')$.*

Proof. Suppose $he = h \frown hr$. Because $h(m).i$ is total, the return event e is well defined. We must now show that he is fence consistent.

- If $m \in \mathrm{dom}\, f$, then because $pi(m, h) \wedge vmf_com(he, f)$ holds, $e \in \mathrm{ran}\, hr$. Furthermore, by Lemma 22, there must exist an hr' such that $\mathrm{ran}(hr') = \mathrm{ran}(hr) \backslash \{e\}$ (i.e., $\langle e\rangle \frown hr'$ is a permutation of hr) and $FC(h \frown \langle e\rangle \frown hr', f')$ holds.
- Otherwise, i.e., $m \notin \mathrm{dom}\, f$, we have

$$\begin{array}{ll} FC(he, f) & \\ \Rightarrow \exists f' \bullet FC(he \frown \langle e\rangle, f') & \text{, by Lemma 21} \\ \Leftrightarrow \exists f' \bullet FC(h \frown hr \frown \langle e\rangle, f') & \text{, definition of } he \\ \Leftrightarrow \exists f'' \bullet FC(h \frown \langle e\rangle \frown hr, f') & \text{, Lemma 22} \end{array}$$ ◀

Compositionality refers to histories of multiple concurrent objects [22]. To formalise this, one must consider histories in which the object corresponding to each event may be distinguished, and hence the event types above must be extended with object names. We assume $e.obj$ returns the object corresponding to event e. For an object z and history h, we let $h_{|z}$ denote the subhistory of h with all events of object z. Prior to our new result that fence consistency is compositional (Theorem 24), we give a new proof of compositionality for linearizability.

▶ **Theorem 24** (Linearizability is compositional). *For any history h, there exists an extension he of h and a mapping function f such that $LIN(he, f)$ if, and only if, for each object z, there exists an extension he_z of $h_{|z}$ and a mapping function f_z such that $LIN(he_z, f_z)$.*

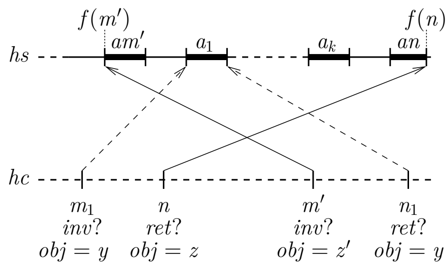


Figure 6 Assumed minimal reordering.

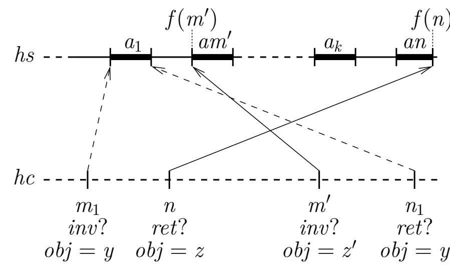


Figure 7 Swapped ordering.

Proof. The only if direction is trivial.

For the other direction, for each object z, suppose $LIN(he_z, f_z)$ holds for some extension he_z of $h_{|z}$ and mapping function f_z. The proof is by contradiction. Suppose that for every extension he of h, and every mapping f to a sequential history, we have:

$$\begin{aligned} & \neg LIN(he, f) \\ = \; & \neg VMF(he, f) \vee \neg lin(he, f) \vee \neg total(he, f) && \text{, by definition} \\ = \; & VMF(he, f) \wedge total(he, f) \Rightarrow \neg lin(he, f) && \text{, by logic} \end{aligned}$$

Thus, we assume $VMF(he, f) \wedge total(he, f)$ and prove $\neg lin(he, f)$.

For $\neg lin(he, f)$ to hold, by definition, there must exist indices n, m' in he such that $n < m' \wedge ret?(he(n)) \wedge inv?(he(m')) \wedge f(n) > f(m')$, i.e., n and m' are indices of operation calls where $he(n)$ returns before $he(m')$, and f reorders these calls when mapping to hs. Suppose $he(n).obj = z$ and $he(m').obj = z'$, and let an and am' be the operation calls corresponding to $he(n)$ and $he(m')$ in hs, respectively. If $z = z'$, we get an immediate contradiction to the assumption that there exists an he_z and f_z such that $LIN(he_z, f_z)$ holds. Therefore, we assume $z \neq z'$.

Now, pick an f such that the number of reordered operation calls that invalidate $lin(he, f)$ are minimal, then pick $n, m' \in \operatorname{dom} f$ such that $f(n) - f(m')$ is minimal and n, m' violate $lin(he, f)$. Because $ret?(he(n)) \wedge inv?(he(m'))$, the smallest possible value of $f(n) - f(m')$ is 3, which occurs if, in hs, the operation call an occurs immediately after am'. However, in this case, because $z \neq z'$, operation calls an and am' commute, i.e., there must exist another valid sequential history in which the order of an and am' are swapped, and we obtain a contradiction to $f(n) > f(m')$. Therefore, assume $f(n) - f(m') > 3$, i.e., some finite number of operation calls $a_1, a_2, \ldots a_k$, occur between $f(m')$ and $f(n)$ in hs.

Consider a_1, and suppose the invocation/return events corresponding to a_1 occur at m_1 and n_1 in he, respectively. We must have $m_1 < n \wedge m' < n_1$, otherwise, we obtain a contradiction to minimality of $f(n) - f(m')$ (see Fig. 6). Let $y = a_1.obj$ be the object corresponding to a_1. We now have two cases.

- Case $y \neq z'$. Because operations of different objects commute, calls a_1 and am' may be swapped in hs, to produce another valid sequential history hs' and mapping f' such that $f'(n) - f'(m') < f(n) - f(m')$, contradicting minimality of $f(n) - f(m')$ (see Fig. 7).
- Case $y = z'$. Here, a_1 and am' may not be swapped, however, we must have $a_2.obj = a_3.obj = \cdots = a_k.obj = z'$, otherwise, we may swap the first a_i such that $a_i.obj \neq z'$ with each of $am', a_1 \ldots a_{i-1}$ to again contradict minimality of $f(n) - f(m')$. However, we now have $a_k.obj = z'$ and $an.obj = z$, i.e., $a_k.obj \neq an.obj$, so a_k and an may be swapped to produce a valid sequential history, giving us our final contradiction. ◀

▶ **Theorem 25** (Fence consistency is compositional). *For any history h, there exists an extension he of h and a mapping function f such that $FC(he, f)$ if, and only if, for each object z, there exists an extension he_z of $h_{|z}$ and a mapping function f_z such that $FC(he_z, f_z)$.*

Proof. The only if direction is trivial.

For the other direction, for each object z, suppose $FC(he_z, f_z)$ holds for some extension he_z of $h_{|z}$ and mapping function f_z. The proof is by contradiction. Suppose that for every extension he of h, and every mapping f to a sequential history, we have:

$$\begin{aligned} & \neg FC(he, f) \\ = & \neg VMF(he, f) \vee \neg sc(he, f) \vee \neg fc_ord(he, f) \vee \neg fc_com(he, f) && \text{, by definition} \\ = & \; VMF(he, f) \wedge sc(he, f) \wedge fc_com(he, f) \Rightarrow \neg fc_ord(he, f) && \text{, by logic} \end{aligned}$$

Assuming $VMF(he, f) \wedge sc(he, f) \wedge fc_com(he, f)$ holds, we attempt to prove $\neg fc_ord(he, f)$. For $\neg fc_ord(he, f)$, by definition, there must exist indices n, m' in he such that

$$n < l < m' \wedge f(n) > f(m') \quad (6)$$

$$ret?(he(n)) \wedge empty?(he(l)) \wedge inv?(he(m')) \wedge he(n).pr = he(l).pr \quad (7)$$

By (6), f reorders operation calls $he(n)$ and $he(m')$, and by (7), n, l and m' are indices corresponding to a return, empty and invocation, respectively and both $he(n)$ and $he(l)$ correspond to the same process. We assume $he(n).obj = z$ and $he(m').obj = z'$, and let an and am' be the operation calls corresponding to $he(n)$ and $he(m')$ in hs, respectively. If $z = z'$, we get an immediate contradiction to the existence of an extension he_z of $h_{|z}$ and mapping function f_z such that $FC(he_z, f_z)$. Therefore, we assume $z \neq z'$.

Pick an f as well as indices n and m' as in Theorem 24. For the minimal value of $f(n) - f(m')$ (i.e., if $f(n) - f(m') = 3$) we obtain a contradiction as in Theorem 24. Therefore, assume $f(n) - f(m') > 3$, i.e., some finite number of operation calls $a_1, a_2, \ldots a_k$, occur between $f(m')$ and $f(n)$ in hs. Consider a_k, and suppose the invocation/return events corresponding to a_k occur at m_k and n_k in he, respectively. We have that $a_k.obj \neq z$ (otherwise we get a contradiction to minimality by swapping a_k and am') and cases:

- If $l < m_k$ or $n_k < l$, because $a_k.obj \neq z$ we obtain an immediate contradiction to the assumption that an and am' are different objects such that $f(n) - f(m')$ is minimal.
- Else if $m_k < n \wedge m' < n_k$, the proof proceeds as in Theorem 24.
- Else if $n < m_k < l$, then $a_k.obj = z$, otherwise we can swap a_k and an to contradict minimality of $f(n) - f(m')$. In fact $a_i.obj = z$ must hold for all $1 \leq i \leq k$. But now $am'.obj \neq a_1.obj$, and hence can be swapped, once again contradicting minimality.
- Finally, if $l < n_k < m'$, we obtain our final contradiction to minimality of $f(n) - f(m')$ using a similar argument to $n < m_k < l$. ◀

5 A Correctness Condition Hierarchy

As we've seen, there are several different correctness conditions that are appropriate for different types of algorithms over different memory architectures. The literature includes many others (e.g., k-linearizability [20], eventual consistency [36], quantitative quiescent consistency [25]), which we have not covered in this paper.

Using our framework, for the conditions we have considered, it is possible to formally establish a hierarchy based on order and commitment properties. We first link ξ-quiescent, fence and flush consistency. Fence and flush consistency are defined on histories that consider slightly different aspects of a system's behaviour. To relate the two conditions, we

consider histories in which writes and flushes as well as buffer empty events are recorded. To this end, we define $Event_\xi ::= Event \mid \xi\langle\langle P\rangle\rangle$, $Event_\phi ::= Event \mid \omega\langle\langle P\rangle\rangle \mid \phi\langle\langle P\rangle\rangle$ and $Event_C ::= Event_\xi \mid Event_\phi$, with the understanding that $\langle \phi(p), \xi(p)\rangle$ is concatenated to the history if the flush $\phi(p)$ causes the buffer of process p to become empty, while $\langle ret(p, op, out), \xi(p)\rangle$ is concatenated whenever $ret(p, op, out)$ occurs and the buffer of p is empty. A history h is *legal* if both $h_{|Event_\phi}$ and $h_{|Event_\xi}$ are legal. This definition of *legal* is also extension closed.

▶ **Proposition 26.**
1. *$FLC \Rrightarrow FC$, but not vice versa, and*
2. *$FC \Rrightarrow QC_\xi$, but not vice versa.*

Proof.
1. The forward direction follows by expanding the definitions of FLC and FC, then using the fact that both $wflc_ord \Rrightarrow fc_ord$ and $wflc_com_1 \wedge wflc_com_2 \Rrightarrow fc_com$. To show the other direction does not hold, consider the following history, where $p, q \in P$ and `enq`, `deq` are enqueue and dequeue operations on a concurrent queue, respectively. The history is clearly a legal fence consistent history, but it is not flush consistent.

$$\langle inv(p, \texttt{enq}, 1), \omega(p), ret(p, \texttt{enq}), inv(p, \texttt{enq}, 2), \omega(p), \phi(p), inv(q, \texttt{deq}), ret(p, \texttt{enq}), \phi(p), \xi(p), ret(q, \texttt{deq}, emp), \xi(q)\rangle$$

2. The forward direction follows by expanding the definitions of FC and WQC_ξ, then using the fact that both $fc_ord \Rrightarrow wqc_ord_\xi$ and $fc_com \Rrightarrow wqc_com_\xi$ hold.
   To show the other direction does not hold, consider the following history, where $q_1, q_2, q_3 \in P$ and `enq`, `deq` are enqueue and dequeue operations on a concurrent queue, respectively. The history is clearly ξ-quiescent consistent, but not fence consistent.

$$\langle inv(q_1, \texttt{enq}, 1), ret(q_1, \texttt{enq}), inv(q_2, \texttt{enq}, 2), \xi(q_1), inv(q_3, \texttt{enq}, 3), ret(q_2, \texttt{enq}), \xi(q_2), ret(q_3, \texttt{enq}), \xi(q_3), inv(q_1, \texttt{deq}), ret(q_1, \texttt{deq}, 3)\rangle$$

◀

It is also possible to link correctness conditions on totally ordered memory with those on TSO memory. It is straightforward to prove the following propositions.

▶ **Proposition 27.** *If h is legal, then for any mapping function f, $LIN(h, f) \Rightarrow FLC(h, f)$.*

One may think of totally ordered memory as being a special case of TSO memory where the buffer is always empty. Here, using the strategy for recording histories of type $History_\xi$ described in Section 4.2, under totally ordered memory, the return for each process p is immediately followed by $\xi(p)$, i.e.,

$$ret_emp(h) \quad \widehat{=} \quad \forall\, n : \mathrm{dom}\, h, p : P \bullet ret_p?(h(n)) \Rightarrow n+1 \in \mathrm{dom}\, h \wedge empty_p?(h(n+1))$$

▶ **Proposition 28.** *If h is legal and $ret_emp(h)$ holds, then for any mapping function f, $LIN(h, f) \Leftrightarrow FC(h, f)$ and $QC(h, f) \Leftrightarrow WQC_\xi(h, f)$.*

One may also think of totally ordered memory as being a type of TSO memory where flushes occur immediately after each write. Here, using the strategy for recording histories of type $History_\phi$ defined in Section 4.4, we obtain the following property, which states that for any process p, a flush for p immediately follows each write by p.

$$imm_fl(h) \quad \widehat{=} \quad \forall\, n : \mathrm{dom}\, h, p : P \bullet \omega_p?(h(n)) \Rightarrow n+1 \in \mathrm{dom}\, h \wedge \phi_p(h(n+1))$$

▶ **Proposition 29.** *Suppose $h \in History_\phi$ is legal and $imm_fl(h)$ holds. Then for any mapping function f, $LIN(h, f) \Leftrightarrow FLC(h, f)$.*

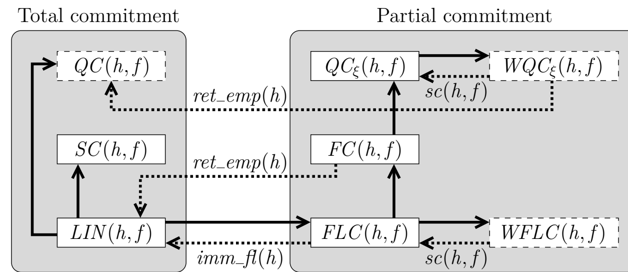


Figure 8 Reltionships between correctness conditions for a history h and mapping function f. The arrows represent implication with dashed versions representing conditional implication; the label on each arrow represents the required condition. The conditions within solid boxes ensure sequential consistency.

An overview of the hierarchy is presented in Fig. 8, where we assume h is a legal history and f a mapping function. Each solid arrow in Fig. 8 denotes implication, and each dashed arrow denotes conditional implication with the condition corresponding to the label. Note that Fig. 8 allows one to easily deduce transitivity properties, e.g., if $FC(h, f)$ and $ret_emp(h)$ hold, then $SC(h, f)$ holds.

6 Conclusions

Correctness of a concurrent object is defined with respect to a correctness condition, which is a relation on its behaviours against those of a sequential specification object. Algorithms implementing such objects must cope with the additional challenges of distributed memory; the low-level effects of write buffers in most modern processors (e.g., x86, ARM) only provide *relaxed memory* guarantees. The end goals of programmers that use concurrent objects and designers that develop algorithm for concurrent objects differ, a large number of correctness conditions have been defined in the literature. We have provided a framework within which these can be formalised, which in turn allows the relative strengths of different conditions to be compared.

Within our framework, we have defined well-known conditions for totally ordered memory (sequential consistency, linearizability and quiescent consistency), as well as newly developed conditions for TSO memory (weak ξ-quiescent consistency and weak flush consistency). To characterise implementations that are also sequentially consistent, we define stricter variations of both conditions that guarantee sequential consistency. We identify and develop a new compositional condition for TSO architectures, fence consistency, which is weaker than flush consistency, but stricter than ξ-quiescent consistency. Notable in our framework is the capability of specifying partial commitment properties, which provide the flexibility needed to cope with delayed operation effects due to relaxed memory.

The study of correctness conditions for concurrent objects under relaxed memory is new [12, 32, 35, 17, 6]. Of these, [12, 17, 6, 35] consider linearizability, but [17, 6] facilitate optimisation (via `fence` removal) by weakening the abstract specification, while [12, 35] weaken definition of linearizability so that the interval of execution for an operation is expanded. We believe flush consistency to be equivalent to the weaker definition of linearizability given in [35], however because Travkin et al. [35] do not provide a formal definition, this is difficult to verify. Jagadeesan et al [24] provide a framework that enables one to develop correctness conditions for many different relaxed memory models by decoupling buffer effects from the correctness conditions at hand, however, they only formalise sequential consistency and linearizability. More recently, Batty et al. [5] have developed methods for proving observational refinement

directly for C11 specifications, which are weaker than TSO. They develop a compositional correctness condition that is stricter than linearizability when C11 is restricted to totally ordered behaviours. Using our framework to formalise their correctness condition to compare them to the conditions in this paper is a subject of further study. Future work will also consider correctness conditions for software transactional memory [19].

This paper has mainly considered correctness conditions from an algorithm designer's perspective. Satisfying the requirements of programmers introduces another dimension to this problem (e.g., [18]). Our work does not differ from, say [17, 6], in that observational refinement is only assured for data independent clients. Extending our results to cope with other real-world issues such as ownership transfer (like [17, 6]) is a subject of future work.

References

1. S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
2. J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7, 2014.
3. H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In T. Ball and M. Sagiv, editors, *POPL*, pages 487–498. ACM, 2011.
4. H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, 1994.
5. M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In R. Giacobazzi and R. Cousot, editors, *POPL*, pages 235–248. ACM, 2013.
6. S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In H. Seidl, editor, *ESOP 2012*, volume 7211 of *LNCS*, pages 87–107. Springer, 2012.
7. D. Chase and Y. Lev. Dynamic circular work-stealing deque. In P. B. Gibbons and P. G. Spirakis, editors, *SPAA*, pages 21–28. ACM, 2005.
8. R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set. In *CAV*, volume 4144 of *LNCS 4144*, pages 475–488. Springer, 2006.
9. J. Derrick, B. Dongol, G. Schellhorn, B. Tofan, O. Travkin, and H. Wehrheim. Quiescent consistency: Defining and verifying relaxed linearizability. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *FM*, pages 200–214. Springer, 2014.
10. J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4, 2011.
11. J. Derrick, G. Schellhorn, and H. Wehrheim. Verifying linearisabilty with potential linearisation points. In M. Butler and W. Schulte, editors, *FM 2011*, volume 6664 of *LNCS*, pages 323–337. Springer, 2011.
12. J. Derrick, G. Smith, and B. Dongol. Verifying linearizability on TSO architectures. In E. Albert and E. Sekerinski, editors, *iFM*, pages 341–356. Springer, 2014.
13. S. Doherty, D. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. Steele Jr. Dcas is not a silver bullet for nonblocking algorithm design. In P. B. Gibbons and M. Adler, editors, *SPAA*, pages 216–224. ACM, 2004.
14. B. Dongol, O. Travkin, J. Derrick, and H. Wehrheim. A high-level semantics for program execution under Total Store Order memory. In Z. Liu, J. Woodcock, and H. Zhu, editors, *ICTAC*, volume 8049 of *LNCS*, pages 177–194. Springer, 2013.
15. I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

16 M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In J. W. Davidson, K. D. Cooper, and A. Michael Berman, editors, *PLDI*, pages 212–223. ACM, 1998.

17 A. Gotsman, M. Musuvathi, and H. Yang. Show no weakness: Sequentially consistent specifications of TSO libraries. In M. Aguilera, editor, *DISC 2012*, volume 7611 of *LNCS*, pages 31–45. Springer, 2012.

18 A. Gotsman and H. Yang. Linearizability with ownership transfer. *Logical Methods in Computer Science*, 9(3), 2013.

19 R. Guerraoui and M. Kapalka. *Principles of Transactional Memory.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

20 T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and Sokolova A. Quantitative relaxation of concurrent data structures. In R. Giacobazzi and R.dhia Cousot, editors, *POPL*, pages 317–328. ACM, 2013.

21 T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In P. R. D'Argenio and H. C. Melgratti, editors, *CONCUR*, pages 242–256. Springer, 2013.

22 M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

23 M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

24 R. Jagadeesan, G. Petri, C. Pitcher, and J. Riely. Quarantining weakness – compositional reasoning under relaxed memory models (extended abstract). In Felleisen M and P. Gardner, editors, *ESOP*, volume 7792 of *LNCS*, pages 492–511. Springer, 2013.

25 R. Jagadeesan and J. Riely. Between linearizability and quiescent consistency – quantitative quiescent consistency. In J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsoupias, editors, *ICALP II*, volume 8573 of *LNCS*, pages 220–231. Springer, 2014.

26 L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

27 F. Liu, N. Nedev, N. Prisadnikov, M. T. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. In J. Vitek, H. Lin, and F. Tip, editors, *PLDI*, pages 429–440. ACM, 2012.

28 S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In T. D'Hondt, editor, *ECOOP 2010*, volume 6183 of *LNCS*, pages 478–503. Springer, 2010.

29 G. Schellhorn, H. Wehrheim, and J. Derrick. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. on Computational Logic*, 2014.

30 P. Sewell, S. Sarkar, S. Owens, F.Z. Nardelli, and M.O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

31 N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.

32 G. Smith, J. Derrick, and B. Dongol. Admit your weakness: Verifying correctness on TSO architectures. In *FACS*. Springer, 2014. To appear (accepted 21 July, 2014).

33 D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence.* Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2011.

34 J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1992.

35 O. Travkin, A. Mütze, and H. Wehrheim. SPIN as a linearizability checker under weak memory models. In V. Bertacco and A. Legay, editors, *HVC*, volume 8244 of *LNCS*, pages 311–326. Springer, 2013.

36 W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

The Love/Hate Relationship with the C Preprocessor: An Interview Study

Flávio Medeiros[1], Christian Kästner[2], Márcio Ribeiro[3], Sarah Nadi[4], and Rohit Gheyi[1]

1 **Federal University of Campina Grande, Brazil**
2 **Carnegie Mellon University, USA**
3 **Federal University of Alagoas, Brazil**
4 **Technische Universität Darmstadt, Germany**

Abstract

The C preprocessor has received strong criticism in academia, among others regarding separation of concerns, error proneness, and code obfuscation, but is widely used in practice. Many (mostly academic) alternatives to the preprocessor exist, but have not been adopted in practice. Since developers continue to use the preprocessor despite all criticism and research, we ask how practitioners perceive the C preprocessor. We performed interviews with 40 developers, used grounded theory to analyze the data, and cross-validated the results with data from a survey among 202 developers, repository mining, and results from previous studies. In particular, we investigated four research questions related to why the preprocessor is still widely used in practice, common problems, alternatives, and the impact of undisciplined annotations. Our study shows that developers are aware of the criticism the C preprocessor receives, but use it nonetheless, mainly for portability and variability. Many developers indicate that they regularly face preprocessor-related problems and preprocessor-related bugs. The majority of our interviewees do not see any current C-native technologies that can entirely replace the C preprocessor. However, developers tend to mitigate problems with guidelines, even though those guidelines are not enforced consistently. We report the key insights gained from our study and discuss implications for practitioners and researchers on how to better use the C preprocessor to minimize its negative impact.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases C Preprocessor, CPP, Interviews, Surveys, and Grounded Theory

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.495

Supplementary Material ECOOP Artifact Evaluation approved artifact available at
`http://dx.doi.org/10.4230/DARTS.1.1.7`

1 Introduction

The C preprocessor is a language-independent tool for lightweight meta-programming that fills a need, among others, for portability and variability. The preprocessor is widely used in practice. It is essentially used in all projects written in C and C++, including many well-known databases and operating systems. In academia, however, the preprocessor has received strong criticism since at least the early 90s. Researchers have criticized its lack of separation of concerns [52, 15, 25, 5, 57], its proneness to introduce subtle errors [58, 33, 15, 40, 44, 6], and its obfuscation of the source code [6, 12, 52, 3, 42]. Additionally, its complexity hinders tool support available in other languages, such as automating refactorings [67, 29, 20, 43, 28, 66]. Many studies have found bugs related to preprocessor use [1, 44, 29, 62, 21, 60]. The C

© Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 495–518

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

preprocessor essentially has not changed since the 70s, but researchers have proposed several alternatives, such as syntactical preprocessors [68, 43, 8], aspect-oriented programming [42, 2] and various forms of metaprogramming. However, such alternatives have not been adopted in practice. Some projects adopt code guidelines such as "code cluttered with `#ifdefs` is difficult to read and maintain, don't do it" in the Linux kernel.[1] Although some tools could enforce such guidelines [40, 56, 6, 60], researchers show that they are not followed strictly in practice [12, 40].

Since developers continue to use the C preprocessor despite all criticism and research, this paper asks the basic question: **How do practitioners perceive the C preprocessor?** Do developers perceive similar problems as researchers indicate, or are problems exaggerated in the research literature? How do developers address potential problems and what kind of alternatives do they seek, if any? Are we possibly faced with a technology-transfer or education problem? Answering such questions provides guidance on research, tool building, technology transfer, and education.

To understand how developers perceive the C preprocessor, we interviewed 40 developers and cross-validated our results with (a) a survey among 202 developers, (b) results mined from software repositories, and (c) prior studies in this field. Complementing prior studies that analyzed how the preprocessor is used in *source code*, we actually talked to developers to solicit *perceptions and opinions*. We focus primarily on conditional compilation, because it is more controversial and error prone than lexical inclusion of files and macro expansion. In our research, we rigorously follow established empirical methods for interviews [17, 36], surveys [11], and text analysis [32]. Specifically, we follow a research method called grounded theory [10, 4]. We report the key insights gained in our interviews (and validated with the survey and other empirical data) and derive implications for practitioners and researchers.

Our results suggest that developers perceive the preprocessor as an elegant solution to handle portability and variability, but they are also well aware of the problems discussed in the academic literature. Developers typically report that they try to avoid preprocessor use or follow code guidelines to minimize problems, such as avoiding `#ifdef` directives inside function bodies and avoiding nesting of conditional directives beyond three levels. Most developers (over 80 %) particularly agree to avoid `#ifdef` blocks that do not align with the code structure (coined undisciplined annotations [40]) because they negatively impact code comprehension, maintainability and error proneness; we detected that actually only few developers introduced 85 % of all such cases.

Developers often deal with bugs related to preprocessor use. Most developers (67 %) agree these bugs are easier to introduce, and 74 % believe that they are harder to detect than other bugs. Our findings suggest that developers use inefficient testing strategies that normally do not detect bugs related to conditional compilation. Open-source developers instead rely on support from end-users to test the source code on different platforms and report bugs.

When asked for alternatives to preprocessor use, developers mainly discussed different encodings, C-language mechanisms, and build-system mechanisms. In contrast, alternatives such as syntactical preprocessors [68, 43, 8] and aspect-oriented programming [42, 2] have not been mentioned by any of our interviewees. Developers argued that new technologies would hinder adoption, since the C compiler is available for many platforms and the preprocessor is always there to deal with variability.

In summary, the main contributions of this paper are:

[1] *Linux* kernel guidelines for patch submission in `Documentation/SubmittingPatches`.

- We interviewed 40 developers to better understand how developers perceive the practical use of the C preprocessor and analyze the results using grounded theory.
- We cross-validated our interview findings by surveying 202 developers and comparing them with results from repository mining and prior studies.
- We discuss results and implications of our study for researchers and practitioners.

2 State of the Art

The preprocessor is widely used in practice, in essentially all projects written in C and C++. It is executed during the compilation process and performs three interacting tasks: (a) it lexically includes files (`#include`), (b) it expands macros (defined with `#define`), and (c) it conditionally excludes part of the source code depending on which and how macros are defined (`#ifdef`, `#if`, etc). All three functions of the C preprocessor have been criticized.

- Lexical inclusion causes large amounts of I/O operations during compilation and slows down the build process. For example, an average file in the *Linux* kernel includes over 300 header files [29]. There is movement in the C community towards a proper build system to replace `#include` directives [23].
- Lexical macros allow all kinds of potential problems [12] since they have no notion of structure, hygiene, or capture avoidance that advanced macro systems support [16, 31, 9, 68, 43]. Developers avoid these problems by following certain patterns when defining macros [12], which are broadly adopted and also checked by a number of static analysis tools [56]. In addition, C++ introduced several language features to replace common uses of preprocessor macros [46, 34].
- Since conditional compilation removes code before compilation, it causes compilers and many other analysis tools to see only parts of the code. It has been criticized as limiting separation of concerns, as obfuscating the code, as being error prone, and as preventing tool support, as we will discuss next. Interactions of conditional compilation with lexical inclusion and macro expansion make it even harder to reason about the preprocessor execution.

In the following, we discuss challenges induced by the C preprocessor (especially conditional compilation, because inclusion and macro expansions are relatively well understood) and available mitigation strategies, as they are discussed in the research literature. These challenges guided us in the design of our study to analyze whether and how the perception of developers differs from that in the research literature.

2.1 Readability and separation of concerns

Many research studies criticized the preprocessor regarding its limited separation of concerns and code obfuscation, which make maintenance and code comprehension difficult [6, 12, 19, 40, 29]. In particular, when conditional directives are used at fine granularity and are strongly scattered, it can be difficult to follow the control flow logic [52, 3]. Such source code is sometimes referred to as the "`#ifdef` hell" by developers [42]. Long and deeply nested conditional compilation directives also can make it difficult to see when specific code fragments are included [52, 33, 5]. Several researchers have proposed aspect-oriented programming as an alternative [42, 2], where optional code would be separated into distinct code artifacts and woven together at compile time, but we are not aware of any adoption beyond some research projects.

```
if (b_ffname != NULL
#ifdef FEAT_NETBEANS
  && netbeansReadFile
#endif
){
  // lines of code
}
```

```
mfp = open(mf_fname
#ifdef UNIX
  , (mode_t)0600
#endif
#if defined (MSDOS)
   , S_IREAD | S_IWRITE
#endif
);
```

```
#if defined (GUI_W32)
void msgNetbeansW32(
#else
void msgNetbeans(Xt client,
#endif
XtInputId *id){
  // lines of code..
}
```

Figure 1 Real code snippets taken from *Vim* with undisciplined annotations.

A specific practice that has been discussed in detail is the use of *undisciplined annotations:* conditional compilation directives that do not align with the code structure as illustrated in Figure 1. Undisciplined annotations have been related to error proneness [12, 40, 44, 29], hindered code understanding and maintainability [6, 12], and limitations in tool support (see below) [18, 19, 6, 51]. An empirical study by Liebig et al. [40] revealed that most conditional compilation directives in 40 open source C projects are disciplined, but 15.6 % of all `#ifdef` blocks do not align with the code structure.

2.2 Combinatorial explosion and parsing unpreprocessed C code

Conditional compilation decides which code fragments to include (including other preprocessor directives) depending on the values of macros. The number of possible preprocessed variants explodes exponentially with the number of macros involved in `#ifdef` and similar directives. C projects often have a large number of conditional directives depending on many macros; for instance, which parts of the *Linux* kernel are compiled depends on more than 12 thousand macros [60, 38].

A separate analysis of every possible preprocessed variant simply does not scale in any but the smallest systems. A typical strategy to cope with the combinatorial explosion is through sampling, for example, by analyzing representative or large variants with most conditional code included. For more systematic sampling, researchers have proposed combinatorial testing strategies [49, 26] and strategies that maximize configuration coverage [60]. Sampling is inherently incomplete though and may not discover issues occurring only in few variants due to interactions or complex `#if` conditions.

Some researchers have started to investigate tools that can parse unpreprocessed code and preserve all compile-time choices during the analysis. While earlier tools used unsound heuristics or supported only specific usage patterns of the preprocessor (e.g., requiring disciplined annotations) [6, 19, 51], more recent tools as *TypeChef* [29, 41] and *SuperC* [22] can accurately parse and analyze unpreprocessed C code, covering all configurations. In the product-line community, such analyses are called *family-based analyses* [64].

2.3 Error proneness and guidelines

Previous studies discussed the error-prone characteristics of the preprocessor [12, 58, 15] and found many bugs related to conditional compilation [1, 21, 60, 61, 44, 30, 12, 53], ranging from dead code to syntax and type errors and to behavioral issues and memory leaks. Spencer and Collyer [58] argue that many macro combinations are tested and often do not even make sense. Others argue that the simplicity of the C preprocessor enables developers to make ad-hoc extensions instead of restructuring the code, which leads to poor code quality and bugs related to preprocessor usage [15, 6].

Code guidelines have been developed to prevent certain common problems, e.g., undisciplined annotations or scope issues with macros [56, 40, 12]. Even though some of them can be enforced automatically by analysis tools [56, 40], research shows that such code guidelines are often but not strictly followed [12, 40].

2.4 Difficulty of developing tool support and syntactic preprocessors

Finally, preprocessor directives also make the development of tool support more difficult [40, 29, 22]. Even simple tasks as removing obsolete macros or identifying dead code require sophisticated analyses [6, 61]. Developing refactoring engines for C code is extremely challenging due to the need to parse unpreprocessed code (possibly with undisciplined annotations) and the need to deal with macro expansion [18, 20, 67, 39]; it is challenging even when conditional compilation is not considered [59, 50].

Many academic proposals for preprocessor alternatives are driven by a desire to provide better tool support and analysis. For example, *ASTEC* is a syntactic preprocessor that enables precise refactoring [43]. Several other syntactic preprocessors or related environments have been proposed [68, 8, 9, 65, 27, 13]. Some researchers propose means to refactor existing C code to alternative implementations [43, 65, 2] or at least undisciplined to disciplined annotations [19, 55, 45]. We are not aware of any adoption of these alternatives in practice though.

All prior studies on the C preprocessor that we are aware of were based on conceptual arguments or evidence extracted from software repositories. Our study is designed to elicit the *perception* of developers by talking to them.

3 Research Method

The goal of our research study is to analyze the strengths, drawbacks, and alternatives to the C preprocessor, as perceived by C developers. We specifically collect information about the C preprocessor that cannot be observed by analyzing only artifacts as in previous studies. We performed this research study primarily by interviewing developers and cross-validating our results with survey questions, other information from software repositories, and related studies. In this section, we give an overview of our research method. Details can be found in the appendix.

For this research, we combine several empirical research methods, including interviews, surveys, and mining software repositories. Empirical research methods allow us to investigate how human developers think and behave. We study not only the outcome of the development process, but assess also their opinions and perceptions. If not conducted carefully, empirical research can result in biased and superficial results. However, whole communities of researchers have investigated how to perform empirical studies that reduce biases and enable reliable and reproducible research despite potentially vague research materials. For example, following strict protocols and documenting steps and research results when analyzing transcribed interviews can mitigate many biases that researchers might otherwise introduce. In addition, cross-validating results from different sources is essential. This way, results complement and confirm each other and form a more reliable bigger picture. In this research, we strictly followed established research methods and cross-validated our results across several sources and with prior research results, as we will explain.

3.1 Research Questions

Our research is motivated by the mismatch between the critique that the C preprocessor has received from academics [6, 12, 19, 40, 29, 58] and the number of alternatives [68, 43, 8, 42, 2] on the one side and the broad use in practice on the other side. Specifically, we raise the following research questions:

RQ1. Why is the C preprocessor still widely used in practice?
RQ2. What do developers consider as alternatives to preprocessor directives?
RQ3. What are common problems of using preprocessor directives in practice?
RQ4. Do developers care about the discipline of preprocessor annotations?

We present the results for each research question separately in Sections 4–7.

3.2 Research Strategy

We performed our research in three phases, designing three studies. In the first phase, we analyzed the literature and identified the research questions stated above (see also Section 2). In the second phase, we performed semi-structured interviews with 40 developers (Study 1). In the third phase, we cross-validated our interview findings by conducting a survey among developers contributing to open source C projects (Study 2; 202 responses), mining data from 24 software repositories (Study 3), and comparing our results with prior research results.

3.3 Corpus

For all three studies, we use a corpus of 24 open source C systems. With the revision history of the systems in the corpus, we identified candidate interviewees and survey participants, and we studied technical aspects. We selected the systems in the corpus based on prior corpus studies on the C preprocessor [12, 38], covering a range of different domains and sizes (2.6 thousand to 7.8 million lines of code). We selected only projects for which we could find developer contact information in commits. The corpus includes the following projects: *Apache, Bash, Bison, Cherokee, Dia, Flex, Fvwm, Gawk, Gnuchess, Gnuplot, Gzip, Irssi, Libpng, Libsoup, Libssh, Libxml2, Lighttpd, Linux, Lua, M4, Mpsolve, Rcs, Sqlite,* and *Vim*.

3.4 Study 1: Interviews

We started our study by interviewing developers on how they perceive the C preprocessor. To reduce any potential bias and to make our study replicable, we followed the established exploratory research method *grounded theory* [10, 4]. We performed *semi-structured* interviews [36, 17], which are informal conversations where the interviewer lets the interviewees express their perception regarding specific topics. To elicit not only the foreseen information, but also unexpected data, we avoided a high degree of structure and formality and, instead, used open-ended questions. To cover the topic broadly, our questions evolved during the interview process based on gained insights [10, 4]. We followed standard guidelines regarding how to perform interviews [36, 17]. For example, we explained the purpose of the interviews, we provided clear transitions between major topics, we did not allow interviewees to get off topic, we allowed interviewees to ask questions before starting the interview, and we scheduled the interviews beforehand.

The interviews were grounded in research questions RQ1–4. We typically started an interview by asking developers about their experience with the C preprocessor and then tried to cover 4-6 different topics. The topics evolved during the interviews, and we asked different topics to different developers based on their background and answers. This is a

standard approach to cover a topic broadly and qualitatively. Questions included *'In which situations do developers use conditional directives?'*, *'How do developers test different macro combinations in their code?'*, and *'What do developers think about directives that split up parts of C constructions?'*; see the appendix for a complete list. In addition to these questions, we used code snippets to ask developers concrete questions about code to encourage them to give more concrete answers. For each interviewee, we searched through the code repositories and selected code snippets related to that specific developer. We sent such snippets by email before the scheduled interview.

We performed 10 phone and 30 email interviews. We initially contacted developers via email presenting some information about our project and asked them to participate. We encouraged developers to perform phone interviews, but we also provided the alternative to answer our questions via email. When necessary, emails interviews involved back and forth conversations (i.e., a dialogue between researcher and participant). We sent at least one additional email with further questions in 19 (63%) out of the 30 email interviews we conducted. This and the fact that we cover the same questions in both phone and email interviews allows us to discuss them together as interviews, and not separately as phone and email interviews. To analyze the interview transcripts, we again followed established research methods: coding the answers, analyzing keywords, organizing them into concepts and categories, and writing memos [10]. We met weekly to discuss the memos and noticed that interviewees progressively started to give similar answers, a situation called *saturation* [10]. At this point, we considered the topic sufficiently clear and focused on other topics that needed further elaboration. The specific coding outcome is listed in the appendix.

We selected participants for the interviews from active developers in the 24 projects of our corpus. By mining the repositories, we identified the top 10 % active developers in each projects that regularly use conditional compilation (ranked by code churn). We sent emails to 213 open-source developers, and 32 (15 %) participated in our interviews. Even though many open-source contributors expressed that they primarily worked in industrial projects, we additionally explored whether interviewees recruited from industrial projects would provide additional insights. After reaching out to our contacts (convenience sampling), eight developers from Brazilian companies accepted to participate in our interviews. Most of our 40 interviewees self-identified as having at least 5 years of experience and many worked both within open-source and industrial contexts. Our developer selection is biased toward developers with experience with conditional complication, which we counteracted however by cross-validating our results with a survey of a broader population. See the appendix for a characterization of the interviewees. In our result presentation, we refer to individual anonymized participants as P1–P40.

3.5 Study 2: Survey

Whereas our interviews are designed to elicit qualitative insights into practices and reasons, our survey is designed to collect quantitative data from a large population. We designed the survey after completing and evaluating the interviews. It is a standard research approach to first perform qualitative investigations to identify relevant questions and subsequently perform a survey to explore them quantitatively in a larger population [24, 47].

With the survey, we explored topics that were unclear from the interviews or where we wanted additional quantitative data. We performed an online survey to reach more developers and again followed common guidelines for that research method [11]. For several questions, the survey included code snippets to make questions more concrete. We mention the survey questions while discussing our results in Sections 4–7. Details on the survey, including the exact questions, can be found in the appendix.

To select participants for our survey, we aimed at reaching a broader audience of developers with different levels of experience regarding conditional directives usage. We randomly sampled from all developers that contributed to the 24 projects in our corpus, excluding our interviewees. We sent emails to 3,091 developers and 202 (6.5%) filled out our survey.

3.6 Study 3: Mining undisciplined annotations

To investigate the issue of undisciplined annotations further, one of the most controversial and criticized issues in the literature, we mined software repositories to analyze different versions of the source code and statically detected undisciplined annotations. Specifically, we analyzed each commit in 14 projects of our corpus. We considered only projects with at least two active developers to compare their programming style regarding undisciplined annotations. We used a modified version of Liebig's *Cppstats* tool [40]. With this tool, we identified all commits that introduced undisciplined annotations, data which we analyzed grouped by developer. Subsequently, we interviewed four developers regarding their reasons for introducing specific undisciplined annotations. See the appendix for details.

3.7 Threats to validity

We selected interviewees by sending email to developers and only those interested in the topic participated in our study. From 40 interviews, even though they cross 24 projects of different sizes and domains and 3 companies, it is difficult to generalize results. Nonetheless, we alleviated these threats by cross-validating with a survey of a larger population. Our survey could be filled out in around 10-15 minutes, which encouraged more developers to participate. Code snippets used in our survey might be misunderstood by developers or might conflate multiple issues; that is, related results can only be interpreted in the context of these snippets. To detect undisciplined annotations, we used *Cppstats* [40], which is based on heuristics and may miss-classify a small number of annotations, but we expect that this does not affect the bigger picture collected across multiple projects.

In the following, we report the main findings of our study, structured by our four research questions. At the end of each research question, we summarize our main findings and the corresponding data sources used.

4 RQ 1: Why is The C Preprocessor Still Widely Used in Practice?

The C preprocessor has been heavily criticized in previous research, which raises the question of why it is still used in practice (RQ1). To fully answer this question, we need two pieces of information. The first is whether developers are actually aware of these (academic) criticisms, and the second is the set of scenarios in which developers find the C preprocessor useful. If developers are aware of the potential problems, but still use the C preprocessor, this suggests that there are cases in which using the C preprocessor is still the preferred or even the only available alternative. However, to identify such cases, we first need to understand the various situations in which the C preprocessor is used.

4.1 Developers' Awareness of C Preprocessor Criticism

We find that developers are aware of the criticism the C preprocessor has received, but they still believe that it is an elegant solution to handle variability and overcome portability problems, if properly used *(P1-P3, P18, P22, P23, P26)*. As one developer (*P39*) explains:

"Every feature of any technology can be abused or misused. When used appropriately, the use of preprocessor directives is not a problem." That said, many developers *(P1-P3, P5-P8, P19, P20, P22-P26, P30-P33)* are aware that they must follow code guidelines to minimize problems related to code comprehension, maintainability, and error proneness (C preprocessor problems are discussed in more detail in Section 6).

4.2 Usage of the C Preprocessor

Our discussion with developers reveals the following fives cases in which they use the C preprocessor.

- *Portability.* Despite being from different domains, many of the systems we studied need to support multiple platforms and operating systems. The C preprocessor is perceived as a convenient way to ensure the system's portability across these different environments. For example, developers often use conditional directives to check settings of operating systems, platforms, compilers, and library versions *(P1-P3, P6, P17-P25, P27).* Based on these settings, developers use certain macros, types, and header files that may only be available when using a specific operating system or compiler. For example, it is not possible to include *Windows* specific headers such as `windows.h` when compiling the source code on *Linux* or *Mac OS*. In addition to handling platform-specific header files, portability also involves checking for specific system constraints as well as making use of platform-specific functionality during implementation *(P1, P3, P18, P19, P21, P24, P27).* For example, in some operating systems, such as *GNU Hurd*, there is no imposed limit on overall file name length, as there is on *Windows*.
- *Variability.* Developers often repeat that they use conditional directives to provide *optional features* or to *select between alternative implementations*. For example, one participant (*P4*) describes his use of variability as follows: *"I use conditional directives to remove parts of the library I do not need, since it makes the binary code much smaller."* Reducing binary size may influence developers' decision in using macros to represent optional functionality (i.e., *features*). The `DEBUG` feature is one extreme example of this, which was mentioned frequently by developers. `DEBUG` is a common feature developers use to print messages along the source code to understand what is going on during execution *(P21, P22, P23).* Since `DEBUG` may not be useful for end-users, developers guard debugging code with the corresponding macro such that end-users can exclude it from the binary code during compilation. Several developers also state that they commonly use conditional directives to support alternate implementations (*P13, P27, P38-P40*). For example, in *Libssh*, developers can choose between different cryptographic libraries such as *Libcrypt* or *Libcrypto*, depending on the characteristics of the cryptographic algorithms they want to use. They find that the C preprocessor provides a convenient way to switch between such libraries at compile-time
- *Code Optimization.* Some developers explain that, apart from excluding unnecessary functionality, they also explicitly use conditional directives to optimize the code for performance or size *(P3, P4, P40).* Interviewees explain that they often do not trust that all compilers will properly optimize their code. Thus, in some cases, developers take the task of optimizing the code into their own hands by implementing known code optimizations after checking for compiler name and version at compile-time using the C preprocessor. For example, the *Gcc* compiler offers some *GNU Extensions* such as type discovery and zero-length arrays. Developers explain that they want to make use of such optimizations if they are aware of their availability as this allows them to actively make the binary code smaller and faster.

```
#ifdef DEBUG
#define DEBUG_MSG printf
#else
#define DEBUG_MSG (format, args...) ((void)0)
#endif
// Developers do not need to check #ifdef DEBUG multiple times..
DEBUG_MSG ("message..");
```

Figure 2 Using function-like macros to avoid avoid code duplication (checking if `DEBUG` is defined multiple times) and to support encapsulation.

- *Code Evolution.* A few developers state that they also often use conditional directives during the introduction of new code versions related to critical functionality *(P27, P28, P39)*. In this context, they introduce new implementations inside conditional directives, but they remove the previous version only when the new version is stable. They explain that by using conditional directives, they can switch between the old and new implementations for testing purposes.
- *Language Limitations.* Several developers mention using conditional directives because of the limitations of the C language *(P6, P14-P16, P20, P36-P38)*. For example, they use `#ifdef` checks to avoid multiple inclusion of header files. Such header guards (or include guards) are probably one of the few applications of the C preprocessor that is accepted by critics [58].

Some developers also mentioned using macros and function-like macros to avoid code duplication and to encapsulate frequently-changing code *(P20, P39, P40)*. This way, developers need to only change the definition of the macro instead of changing all occurrences in the code. For example, Figure 2 shows how function-like macros can be used to define the behavior of `DEBUG_MSG`. While avoiding duplication and supporting encapsulation are not specific to the C language, using the C preprocessor is perceived as a convenient way to change function definitions at compile-time instead of at run-time. Previous studies [46, 35] considered the replacement of preprocessor macros with new features and idioms in the C++ programming language.

4.3 Discussion

We observed that the answers in our interview data reached a saturation point that is why we did not include this research question in our survey. This is also supported by the fact that many of the cases of C preprocessor usage we find (apart from the rare case of supporting code evolution) align with those found in previous work. For example, Ernst et al. [12] observed that portability accounts for 37 % of the use of conditional directives in the systems they examined, while include guards account for 6.2 %. They also found frequent usage of inline functions or function-like macros. Ernst el al. also argue that in order to eliminate some of the preprocessor usage, developers must be confident that the compiler will perform the necessary code optimizations. Our interviews support this and further suggest that, even after more than a decade, developers still lack this confidence in compiler optimizations.

SUMMARY 1

Developers are aware of the criticism the C preprocessor receives, but still use it in the following situations: (1) supporting portability, (2) supporting variability, (3) providing code optimizations, (4) supporting code evolution, and (5) overcoming language limitations.

Data Sources: Interviews (Study 1) and Prior studies [12, 40, 54, 58]

5 RQ 2: What do Developers Consider as Alternatives to Preprocessor Directives?

We have seen that developers are aware of the problems and risks of using the C preprocessor but still have several use cases for which they need the C preprocessor's functionality. While researchers have proposed alternatives [68, 43, 8, 9], we wanted to see which alternatives developers are aware of or would recommend. We asked whether developers have thought about alternatives to the C preprocessor. We did not ask about specific alternatives or tools, because it was apparent that they usually would not be familiar with them. When we asked for preferences, we were only comparing different ways of using the C preprocessor. Our main goal with this question was to identify perceived alternatives, not to judge existing ones.

We generally received three kinds of answers: suggestions to use the C preprocessor in specific ways (guidelines on how to structure the code), suggestions to use in-language runtime mechanisms instead of compile-time mechanisms of the preprocessor, and arguments that the preprocessor cannot be replaced. Equally important is that none of our interviewees mentioned alternative preprocessors, aspect-oriented programming, or other metaprogramming solutions suggested by researchers. In the following, we discuss the three kinds of answers we received, cross-validated with survey findings.

5.1 Guidelines for Structuring Code

The first common suggestion to avoid using the C preprocessor is to separate alternative and optional code on the function, file and directory structure level *(P3, P8, P9, P14, P18, P24)*. For functions, the idea is to define alternative implementations of a function in separate files and to use the build system and the linker to choose the desired one. Figure 3b shows an example of this. Similarly, grouping related files in the same directory can also move compilation control to the build system, i.e., the whole directory will be compiled or not. Such structuring of the code means that no preprocessing is necessary within files. Additionally, the code structure is portable and requires no special tools. Nonetheless, one developer *(P15)* cautions that structuring the code in this way may leave it more difficult to comprehend. It is also difficult to deal with similar functions and code duplication if developers do not use helper functions for the common code.

The answers we received align with previous academic discussions [48, 58], but did not reach a saturation point in our analysis. Therefore, we asked a larger population of developers whether they prefer this code structuring strategy. In the survey, we present the two equivalent code snippets shown in Figure 3, asking developers which one they prefer based on a five-point Likert scale. We find that 30 % prefer to use conditional directives inside function bodies (i.e., Figure 3a), while 60% prefer to use different functions to solve portability concerns (i.e., Figure 3b). The remaining 10% of respondents had no preference between both options.

5.2 Alternative In-language Runtime Mechanisms

Another alternative that was suggested frequently during our interviews is the use of run-time variability binding (i.e., C `if` Statements) instead of compile-time binding with the preprocessor (i.e., `#ifdef` directives). An example of this is shown in Figure 4b.[2] Many

[2] While the decision here is still made statically, it could also be loaded from command-line options.

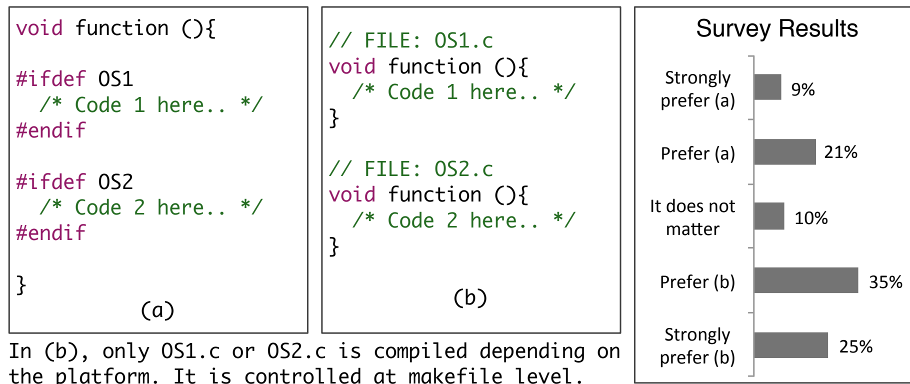


Figure 3 (a) Using preprocessor conditional directives inside function bodies to solve portability concerns. **(b)** Using different functions to avoid conditional directives inside function bodies.

developers state that they prefer to solve variability at run-time, when possible, since it is more flexible *(P1-P4, P6, P23, P40)*. One developer (*P23*) illustrates on this, saying: *"If something can reasonably be done without the preprocessor, I choose [to do it] that way. [Once the binary is there,] it is much more flexible to enable functions at run-time or with a configuration file than having to recompile the project again."*

To achieve such run-time variability, interviewees suggest using variables and enumerators instead of macros with constant values. They also suggest using inline functions to optimize the source code instead of function-like macros. However, developers also caution that run-time variability, e.g., the use of global variables and enumerators, is not thread-safe in C, and that using run-time variability is not possible in some cases. For instance, when runtime checks are not feasible due to performance reasons. One developer clarifies that run-time checks would cause performance overheads when checking for debugging mode, for example. This developer explains that when your goal is to process millions of I/O operations in the Linux kernel, for example, having run-time (i.e., C `if`) debug checks to verify certain assumptions would prevent you from scaling. However, developers still need a mechanism to easily verify assumptions when checking for code correctness, and they suggest that this can be cheaply achieved using the C preprocessor at compile-time.

Since developers' comments about run-time checks were not entirely consistent, we use the survey to see the preference of a broader population. This time, we present the two code snippets shown in Figure 4. In Figure 4a, we use conditional directives, while in Figure 4b, we use run-time variability with C `ifs`. We again ask survey participants to indicate which style they prefer using. Surprisingly, 75 % mentioned that they prefer to use conditional compilation directives (i.e., Figure 4a) while only 19 % prefer to use run-time variability (Figure 4b). The remaining 6 % of developers did not have a preference.

Based on the results of our interviews, we expected a higher percentage of developers to prefer using run-time variability in the survey. Accordingly, we went back to our interview data to see if we can find supporting reasons for why this might be the case. We find that developers might be inclined to use `#ifdefs` instead of `if` checks because of the following reasons. First, as stated by developer *P1*, when using conditional directives, it is easier to see the optional code. In other words, it is clear that the block of code from lines 3 to 8 is optional in Figure 4a. Second, developers *P3* and *P4* mentioned that by using variables instead of macros, developers do not know whether the compiler will optimize the source code. For instance, if the developer does not define `PM3D` in Figure 4b, variable `PM3D_RT`

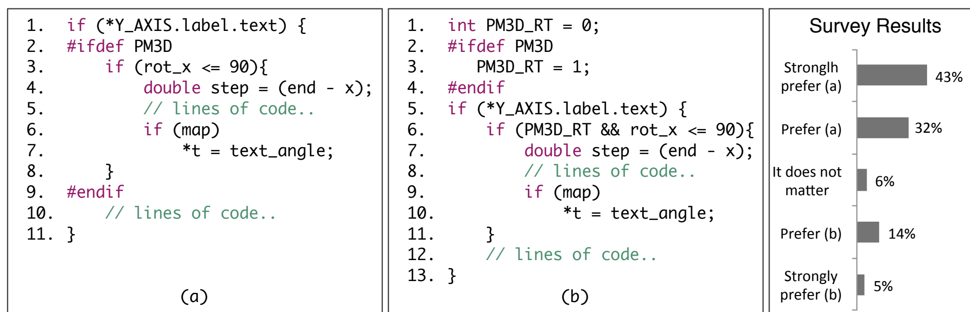


Figure 4 (a) Code snippet of *Gnuplot* with preprocessor conditional directives within function bodies. **(b)** Using local variables to avoid checking preprocessor macros throughout the code.

is always zero, i.e., `false`, and the block of code from lines 7 to 10 becomes unreachable. Developers argue that compilers may perform optimizations to remove such dead code, but there are no guarantees (i.e., this is compiler specific and depends on the compiler settings). Additionally, a few developers mention that some compilers may issue warnings about such unreachable code or about cases where the `if` condition would always be true which they find annoying *(P3, P29)*.

5.3 No Perceived Alternatives

Several developers mention that they have not thought about alternatives to the C preprocessor *(P17, P19-P21, P25-P28)* and that they are comfortable with using the C preprocessor for the purposes previously discussed. As stated by one developer (*P40*): *"Preprocessor directives can be used to remove the most tedious and error-prone parts of programming. It [is] also the only C-native way to conditionally compile when run-time checks are unacceptable [due] to performance [overheads]. There are no alternatives to the C preprocessor for this type of usage without using some tool outside the language."* Similarly, additional developers mentioned that in some cases, they really need to remove parts of the source code *(P1, P6, P23, P27)*. Otherwise, the code will not compile because of platform-specific parts that have not removed. This leads them to argue that it does not matter which alternative one comes up with, one will need the C preprocessor at some point for such a platform-specific conditional compilation problem. Additionally, developers expressed concern that new technologies that replace the C preprocessor are likely not going to be present in all compilers *(P1, P6, P20)*. This shows hesitation to adopt third-party tools or alternate technologies (e.g., aspect-oriented programming [2, 42] or new macro languages such as ASTEC [43]) because of portability concerns.

Summary 2

Developers do not see any current technologies that can entirely replace the C preprocessor. However, some developers routinely use alternate coding styles such as dividing functionality into separate files or functions (preferred by 60 %) and using run-time checks instead of `#ifdef` *checks (preferred by 19 %).*

Data Sources: Interviews (Study 1), Survey (Study 2), and Prior studies [48, 58]

6 RQ 3: What are Common Problems of Using Preprocessor Directives in Practice?

We now try to understand what problems, if any, do developers face while using the C preprocessor. We find that developers' comments generally align with the problems raised in the research literature. Specifically, developers mention the following problems: (1) dealing with preprocessor-related bugs, (2) testing an exponential number of configurations, and (3) difficulty with understanding code with too many `#ifdefs`.

6.1 Preprocessor-related Bugs

A number of developers confirm that bugs related to conditional compilation occur frequently *(P7, P18, P23, P35-P37)* or at least sometimes *(P8-P10, P13, P14, P27)*. Our interviewees list different types of bugs related to the use of preprocessor directives, such as: incompatible macro selection *(P17)*; macros resulting in erroneous control flow *(P20, P22, P26)*; incorrect macro expansion *(P9)*; misspelled macro names *(P22, P23)*; missing variables and functions such as defining a variable in optional code and using it in mandatory code *(P13)*; type errors *(P8, P18, P23)*; syntax errors like missing control flow tokens, e.g., opening and closing brackets *(P24)*; linking problems *(P24)*; behavioral changes due to macro interactions *(P1, P9)*; memory and resource leaks, memory corruption, and race conditions *(P14)*; and incorrect use of `#else` and `#elif`. For instance, `#else` clause incorrectly treating some configurations, or use of `#elif` without an `#else` at the end to treat the default case *(P14)*.

Some interviewees *(P3, P20, P27)* argue that it is hard to deal with a high number of different macro combinations, which may ease the introduction of bugs. Developer *P1* points out that code that does not compile is easy to deal with, but the runtime bugs are the harder ones to detect. On the other hand, some developers *(P8, P10, P13, P20)* mention that even to detect simple compiler errors, someone has to compile the source code using the specific configuration that contains errors which is not that easy. The types of bugs developers mentioned align with various results from previous studies [7, 14, 37, 61, 62, 44] that we discussed in more detail in Section 2. Additionally, the fact that dealing with macro combinations is one of the sources of bugs is consistent with the findings of Iago et al. [1].

Since the data from our interviews is qualitative, we used our survey to quantify the frequency of C preprocessor-related bugs. We also asked developers about the difficulty of introducing preprocessor-related bugs when compared to other types of bugs as well as difficulty of detection. We present the results in Figure 5a-d, which can be summarized as follows. Even though developers believe that preprocessor-related bugs do not occur very frequently (Figure 5a), they find that they are slightly more critical than non-prerprocessor related bugs (Figure 5b) and that they are easier to introduce (Figure 5c) and harder to detect (Figure 5d). Our survey findings are consistent with our prior work [29] that parsed all code of a *Linux* kernel (release `x86`, only) and did not find any syntax errors. On the other hand, when the same authors analyzed *BusyBox*, they only found a few type and linker errors that they reported to developers and which were fixed in subsequent releases [30]. This supports our findings that such types of errors may be rare, but are still important to fix nonetheless.

Both our interview and survey results suggest that, similar to researchers [29, 40, 1], practitioners perceive the C preprocessor as error-prone. However, developers did not mention having any tools that help them with avoiding such errors. Our findings suggest that we need further research on developing tool support to minimize bugs related to preprocessor usage and finding ways to make such tools attractive for developers to use.

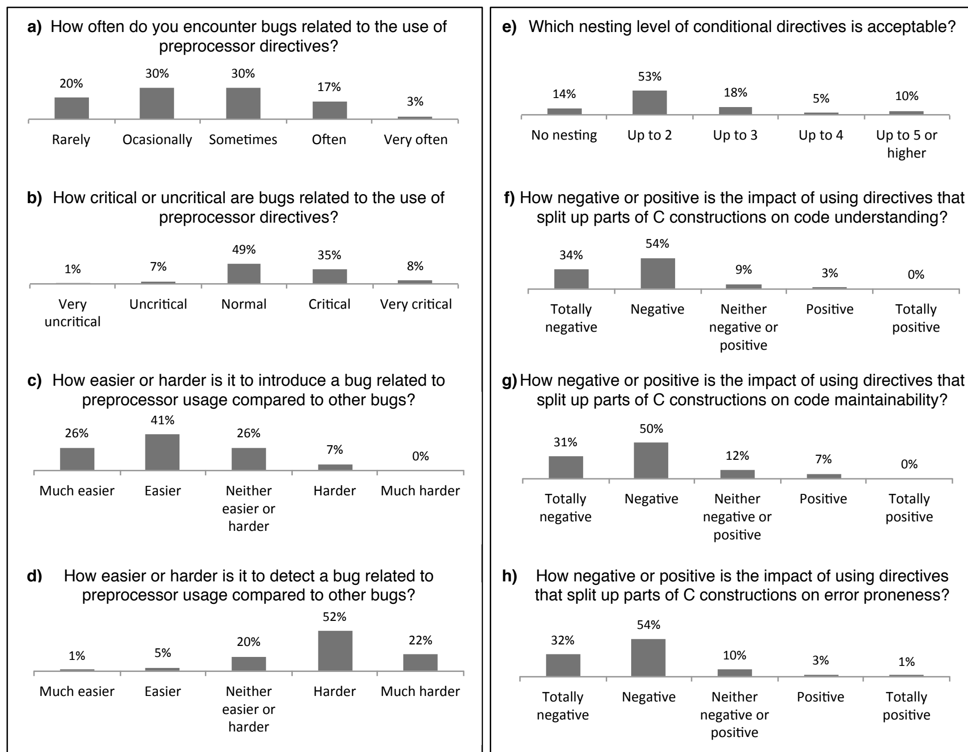


Figure 5 Results of our survey to quantify some findings of our interviews.

6.2 Combinatorial Testing

Developers explain that another problem with using the C preprocessor is dealing with combinatorial explosions. As mentioned by developer *P19*, the use of conditional directives makes the code hard to test and debug since it increases the testing matrix. The number of configurations to test grows exponentially when developers add new preprocessor macros in `#ifdefs`. Assuming that there are n optional and independent macros, developers have 2^n different configurations. Furthermore, developers explain that they also need to consider different compilers, operating systems, and platforms, which is time-consuming and makes automation difficult. For these reasons, many developers *(P1-P6, P9, P17, P19, P20, P22, P23, P25)* mentioned that they normally do not test all different macro combinations due to time and resource constraints. They explain that they do not have an easy way to test all possible combinations.

Many developers *(P9, P17, P19, P20, P22, P23, P25)* mention that what happens in practice is that they check only a few configurations of the source code. Furthermore, some developers *(P11, P18, P24)* check only the default configuration on their own machine with all optional macros active. However, a few developers *(P1, P37)* mention that they additionally consider different platforms besides their own. They say that by compiling the source code with two or three different compilers and using 32 and 64-bit platforms, they are comfortable enough that the code is portable. Similarly, some developers *(P19, P20, P22)* said that they select specific configurations to test by setting different macro combinations manually.

This variation in testing style tells us that there is no systematic way to fully test such systems. We find that developers *(P2, P26)* often rely on end-users to test the source code

using different platform configurations. Developer *P26* explains this as follows: *"I check whatever combinations I can, and some combinations can only be tested on systems to which I have no access. I rely on others to help out or just cross my fingers."* Developer *P2* echoes this, also stating that he heavily relies on his user base to report back errors. This result aligns with a recent study on testing in the *Eclipse* platform by Greiler et al. [24]. Some developers *(P4, P10, P13)* realize that they use a narrow testing strategy and perceive it as a problem, expecting to find additional bugs if they are able to test more configurations. For example, one developer (*P26*) tell us: *"I do not find bugs related to preprocessor usage by running tests, but when running the tests with different combinations of macros."*

We find that testing in industry and open-source projects are different. While our open-source interviewees repeatedly mention testing only a few configurations and relying on user testing, industry developers *(P31, P32, P38)* mention that they test the source code on all supported platforms with all macros active. Additionally, some industrial developers *(P33-P36)* state that they check all combinations and platforms supported. This difference can be explained by the lack of community involvement in the industry context and that the number of used configurations tends to smaller (companies can restrict the number of supported configurations).

To overcome some of the challenges above, several developers *(P8-P12, P14, P31-P34)* mention that they use style checkers and static-analysis tools that often help them avoid bugs. This is especially true in industry projects. Our interviewees used the following tools: *Checkpath*, *Vera++*, *Coverity*, *Cppcheck*, *Valgrind*, *Coccinelle*, and *Lint*. Other developers *(P7, P13)* mentioned that they use at least *Gcc* with all warnings active. However, these tools consider only one configuration at a time, after preprocessing. Thus, these tools do not focus on bugs related to preprocessor usage. Some tools, e.g., *Cppcheck*, try to deal with many configurations by activating one macro at a time and performing the analysis several times, which is time-consuming. *Coccinelle* also handles variability to some extent by building a control-flow graph per function, where statement-level `#ifdefs` are taken into consideration. During the interviews, only one *Linux* developer (*P9*) mentioned a research tool, *Undertaker* [63], that can detect dead `#ifdef`-guarded blocks. However, developers did not mention any of the research tools that could analyze all configurations, such as *TypeChef* [29] or *SuperC* [22].

6.3 Code Comprehension

Our interviews suggest that many developers find it hard to understand code that is filled with `#ifdefs`. Developers *(P1, P5)* mentioned that the mixing of C code and directives interrupts the code logic since they are two independent languages. Developers *(P1, P5, P6, P19, P21, P25, P26)* believe that this mixing can obfuscate the source code making it harder to read, comprehend, and maintain since it is difficult to determine which parts of the code are going to be compiled under which conditions. For example, some developers *(P1, P5, P23, P24)* complain about the use of fine-grained directives within function bodies. It requires the analysis of control flow structures (`if`, `while`, `switch`, and `goto` statements) as well as `#ifdef`, `#ifndef`, `#if`, `#else`, and `#elif` directives. In addition, it becomes harder to understand the control flow, more difficult to check whether opening and closing brackets match, increases code complexity, and may lead to bugs. Additional developers *(P10, P18, P20, P24)* confirm this, saying that they often avoid preprocessor directives because of readability problems. One developer (*P6*) specifically comments on this, saying *"My main problem is that [if] there are macros 7 layers deep[,] I don't understand them."*

We used our survey to gain further insight into the impact of C preprocessor directives on code comprehension and maintainability as shown in Figure 5e. We find that 14 % of

```
1. if (user_callbacks == NULL) {
2. #ifdef HAVE_PTHREAD
3.   callbacks=&ssh_pthread;
4. }
5. #else
6.   return SSH_ERROR;
7. }
8. #endif
                (a)
```

```
1. if (user_callbacks == NULL) {
2. #ifdef HAVE_PTHREAD
3.   callbacks=&ssh_pthread;
4. #else
5.   return SSH_ERROR;
6. #endif
7. }
                (b)
```

Figure 6 (a) Undisciplined annotation. **(b)** One possible equivalent/disciplined version.

our interviewees state that they prefer to avoid nesting preprocessor conditional directives altogether, 53 % do not mind using nesting up to level 2, and only 18 % find that three levels of nesting is still acceptable. Note that only a few developers find that deep nesting levels (i.e., those beyond three) are acceptable. Overall, implicitly, 85 % of the developers see that nesting levels beyond three should be avoided. This aligns with previous work that finds that the average nesting level across 40 analyzed C systems is approximately 1 [38].

Summary 3

Developers face three preprocessor-related problems: (1) preprocessor-related bugs (do not appear often, but are perceived as more critical than other bugs), (2) combinatorial testing (conditional directives increase number of configurations to test), and (3) code comprehension (due to deep nesting of `#ifdefs`*).*

Data Sources: Interviews (Study 1), Survey (Study 2), and Prior studies [38, 29, 40, 1, 7, 14, 37, 61, 62, 44]

7 RQ 4: Do Developers Care About the Discipline of Preprocessor Annotations?

The feasibility of introducing undisciplined annotations (see Section 2) is one of the most criticized aspects of the C preprocessor [12, 40, 44, 29, 6, 22, 18, 19], which is why we dedicate a separate research question for it. Our goal is to find out whether developers also view undisciplined annotations as problematic.

The majority of interviewees *(P1, P5, P17-P28, P32)* agree that the use of preprocessor directives to encompass individual tokens or parts of C syntactical units impacts the quality of code negatively. Such developers emphasize that they would not use undisciplined annotations because they hinder source code readability *(P5, P17, P18, P22, P25, P26, P28)*, obfuscate control flow *(P1, P24)*, and make the code difficult to evolve and maintain *(P20, P22, P23)*. One developer (*P20*) elaborates on this, saying: *"I avoid this kind of directives, they make the source code hard to understand and maintain. My gut feeling keeps screaming possible bugs when I'm faced with a code like that."* Along the same lines, one of these developers recommends to discourage or disallow undisciplined annotations through code guidelines to avoid the aforementioned problems *(P26)*. Another developer *(P30)* stated that code guidelines are important for the homogeneity of the project and that he often asks contributors to rewrite patches to follow the guidelines.

Despite the criticisms we received from most interviewees as shown above, some developers *(P4, P22, P31)* mention that they would still use undisciplined annotations in very specific cases. In such cases, they would also document the code to let others understand their reasoning. Furthermore, some developers (*P5, P7*) are reluctant to change undisciplined annotations once they exist. For instance, one developer (*P5*) states that: *"One thing is to*

not fix what is not broken. The problem is that to refactor a code, you have to understand [it]. If you do not understand [it], it is not easy to refactor. Many developers would say: I am not going to touch that." Developer *P39* mentioned that while he believes it is good to fix undisciplined annotations, it has a very low priority. It is worth noting that none of the developers mentioned using tools to enforce disciplined annotations or identified a lack of tool support in general.

To generalize our findings, we use the survey to quantify the impact of undisciplined annotations on code understanding, maintainability, and error proneness as shown in Figure 5f-h. Our results show that developers generally agree that undisciplined annotations have a negative impact on code understanding (88 %), maintainability (81 %), and error proneness (86 %). However, in a previous study, Schulze et al. [55] could not establish significant differences between disciplined and undisciplined annotations from a program comprehension perspective in a controlled experiment. Nevertheless, they observed that finding and correcting errors is a time-consuming and tedious task in the presence of preprocessor annotations. Additionally, although developers see undisciplined annotations negatively, other researchers [40] detected that almost 16 % of conditional directives are undisciplined annotations.

To investigate this gap between developer preferences and perceptions and the reality in the code base, we performed an additional analysis of software repositories to identify the reasons why developers introduce undisciplined annotations. By analyzing 14 software repositories of our corpus, we detected that only 21 (7 %) out of 299 developers introduced almost 85 % of all undisciplined annotations we found in the software repositories. When we tried to contact these developers, some got defensive and excused their use of undisciplined annotations. For instance, one developer argues that, *"The code was actively rewritten at the time, and it often happens that first drafts of an idea ends up in poor code."*

Figure 6a presents an example of an undisciplined annotation introduced in one of the C projects we examined. When we discussed this code fragment with its author, the author mentioned to prefer the equivalent code snippet in Figure 6b as a replacement. Another developer who we contacted about undisciplined annotations stated to use such annotations to avoid cloning the source code as well as compiler warnings. Figure 7a presents the undisciplined annotation introduced by this developer. When discussing the code, the developer showed us the alternative in Figure 7b that clones the source code (see lines 5 and 8) and another that generates compiler warnings as shown in Figure 7c, both of which seemed unacceptable to that developer. In this latter case, variable `failed` is always `true` when macro `USE_NTLM_AUTH` is not defined. In addition, this developer mentioned that since the undisciplined annotation in the original code was not repeated in many places, this minimizes potential problems. Such examples show that there may be situations where developers would prefer to use undisciplined annotations. In a previous study, We proposed alternatives to discipline annotations without cloning the source code [45]. However, they did not take compiler warnings into consideration. According to our data, compiler warnings seem to be a problem that may hinder the adoption of syntactical preprocessors despite of the existence of compiler attributes to ignore specific warnings.

SUMMARY 4

Most developers agree that undisciplined annotations impact code understanding, maintainability, and error proneness. However, there are cases where developers use undisciplined annotations to avoid code clones and compiler warnings.

Data Sources: Interviews (Study 1), Survey (Study 2), Mining Repositories (Study 3), and Previous work [45, 55, 40]

```
1. #ifdef USE_NTLM_AUTH
2. if (priv->sso_available) {
3.   conn->state = SSO_FAILED;
4. } else {
5. #endif
6.   conn->state = NTLM_FAILED;
7. #ifdef USE_NTLM_AUTH
8. }
9. #endif
            (a)
```

```
1. #ifdef USE_NTLM_AUTH
2. if (priv->sso_available) {
3.   conn->state = SSO_FAILED;
4. } else {
5.   conn->state = NTLM_FAILED;
6. }
7. #else
8. conn->state = NTLM_FAILED;
9. #endif
            (b)
```

```
1. boolean failed = TRUE;
2. #ifdef USE_NTLM_AUTH
3. if (priv->sso_available) {
4.   conn->state = SSO_FAILED;
5.   failed = FALSE;
6. }
7. #endif
8. if (failed){
9.   conn->state = NTLM_FAILED;
10.}          (c)
```

Figure 7 (a) Undisciplined annotation. **(b)** Alternative that clones code. **(c)** Alternative that generates compiler warnings.

8 Concluding Remarks

We performed interviews with 40 developers about their perceptions of the C preprocessor, used grounded theory to analyze the data, and cross-validated the results with data from a survey with 202 developers, repository mining, and previous studies. Our study makes a step toward understanding how developers perceive the C preprocessor, enabling new perspectives on practices, guidelines, tools and technology transfer, and possible research directions. We found that the C preprocessor is still widely used in practice mainly to solve portability and variability problems. Developers are aware of problems and follow guidelines and adopt runtime variation, but they see no alternatives to the preprocessor; developers are skeptical about using new technologies outside the language. In addition, preprocessor-related bugs are perceived as less frequent, but easier to introduce, harder to fix, and more critical than other bugs. We also find that more than 80 % of developers do not like to use undisciplined annotations because they negatively impact code maintainability, comprehension and error proneness.

Our study has several implications for practitioners and researchers:

1. **Guidelines and enforcement.** Practitioners use guidelines to avoid common well-known pitfalls of the C preprocessor. In addition to information mined from repositories [12, 40, 44], our study provides a first step to develop guidelines grounded in data and taking into account developer preferences and acceptance. For example, we found strong evidence that developers support separating portable code at functions or files granularity, avoiding deep nesting of `#ifdef` directives, and avoiding undisciplined annotations. Whereas many developers routinely use analysis tools and respond to their warnings, there are currently few tools to enforce preprocessor-related guidelines systematically.
2. **Quality assurance.** Configurations are rarely tested systematically or even exhaustively, and developers perceive this as a problem. Restriction on configurations (especially in industrial projects) and community involvement (especially in open source projects) are current pragmatic strategies to cope with large configuration spaces (similar to plug-in systems [24]). We argue that systematic sampling [49, 60] and family-based analyses [64] are promising directions.
3. **Tool design and technology transfer.** The cross-platform availability with every compiler is an asset of the C preprocessor that developers are not willing to give up and that limits the adoption of alternative tools. Changes at the standardization level seem more effective, as done with module systems [23] and C++ extensions to replace the need for lexical includes and macros [46, 34]. Researchers might want to make their tools more attractive to developers by taking their perspective and needs into account. Low awareness of research tools indicates a communication problem. External tools that automatically detect guideline violations (such as undisciplined annotations) and propose

fixes (e.g., refactorings) can likely have a larger impact on practice and simplify work for downstream analysis or refactoring tools that can take advantage of limited usage patterns.

The purpose of this study is not to design new language mechanisms, but we discussed preferable alternatives, many of which have been adopted in modern languages. Still, the large amount of existing C code (and also Fortran and C++ code which frequently use the C preprocessor) and the difficulty of analzying and porting it due to the particularities of the C preprocessor make it worth studying the problem both from a technical perspective and also as a technology transfer problem. While the results do not directly generalize beyond the C preprocessor, it demonstrates perceptions of developers that might be of interest also to other language and tool designers.

Acknowledgements. We thank all participants of our interviews and surveys for their time and open discussions. We are grateful for constructive feedback from Sven Apel, Jörg Liebig, Janet Siegmund, and Iago Abal on earlier drafts of this paper. This work was supported by CNPq grants 573964/2008-4 (INES), 306610/2013-2, 477943/2013-6 and 460883/2014-3, NSF grant CCF-1318808, NSERC CGS-D2-425005 and the DFG Project E1 within CRC 1119 CROSSING.

References

1 Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 Variability bugs in the Linux Kernel: A qualitative analysis. In *Proceedings of the International Conference on Automated Software Engineering*, ASE. IEEE/ACM, 2014.

2 Bram Adams, Wolfgang De Meuter, Herman Tromp, and Ahmed E. Hassan. Can we refactor conditional compilation into aspects? In *Proceeding of the International Conference on Aspect-Oriented Software Development*, AOSD. ACM, 2009.

3 Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The evolution of the Linux build system. *Electronic Communications of the European Association for the Study of Science and Technology*, 2008.

4 Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16(4), 2011.

5 Michalis Anastasopoules and Critina Gacek. Implementing product line variabilities. In *Proceedings of the Symposium on Software Reusability*, SSR. ACM, 2001.

6 Ira Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Procedings of the Working Conference on Reverse Engineering*, WCRE. IEEE, 2001.

7 Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. In *Proceedings of the International Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA. ACM, 2008.

8 Quentin Boucher, Andreas Classen, Patrick Heymans, Arnaud Bourdoux, and Laurent Demonceau. Tag and prune: A pragmatic approach to software product line implementation. In *Proceedings of the International Conference on Automated Software Engineering*, ASE. ACM, 2010.

9 Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of the Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM. ACM, 2002.

10 JulietM Corbin and Anselm Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13(1), 1990.

11 Don A. Dillman, Jolene D. Smyth, and Leah Melani Christian. *Internet, Phone, Mail, and Mixed-Mode Surveys: The Tailored Design Method*. Wiley, 2014.

12 Michael Ernst, Greg Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12), 2002.

13 Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Transaction on Software Engineering and Methodology*, 21(1), 2011.

14 David Evans. Static detection of dynamic memory errors. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI. ACM, 1996.

15 Jean-Marie Favre. Understanding-in-the-large. In *Proceedings of the International Workshop on Program Comprehension*, IWPC, 1997.

16 Matthew Flatt. Composable and compilable macros:: You want it when? In *Proceedings of the International Conference on Functional Programming*, ICFP. ACM, 2002.

17 Uwe Flick. *An Introduction to Qualitative Research*. SAGE Publications, 2014.

18 Alejandra Garrido and Ralph Johnson. Challenges of refactoring C programs. In *Proceedings of the International Workshop on Principles of Software Evolution*, IWPSE, 2002.

19 Alejandra Garrido and Ralph Johnson. Analyzing multiple configurations of a C program. In *Proceedings of the International Conference on Software Maintenance*, ICSM. IEEE, 2005.

20 Alejandra Garrido and Ralph E. Johnson. Embracing the c preprocessor during refactoring. *Journal of Software: Evolution and Process*, 25(12), 2013.

21 Brady J. Garvin and Myra B. Cohen. Feature interaction faults revisited: An exploratory study. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE. IEEE, 2011.

22 Paul Gazzillo and Robert Grimm. SuperC: parsing all of C by taming the preprocessor. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI. ACM, 2012.

23 Doug Gregor. A module system for the C family, 2012. Remarks by Doug Gregor at The sixth general meeting of LLVM Developers and Users.

24 M. Greiler, A. van Deursen, and Margrete-Anne Storey. Test confessions: A study of testing practices for plug-in systems. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2012.

25 Ying Hu, Ettore Merlo, Michel Dagenais, and Bruno Laguë. C/C++ conditional compilation analysis using symbolic execution. In *Proceeding of the International Conference on Software Maintenance*, ICSM. IEEE, 2000.

26 Martin Fagereng Johansen, Oystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the International Software Product Line Conference*, SPLC, 2012.

27 Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the International Conference on Software Engineering*, ICSE. ACM, 2008.

28 Christian Kästner, Sven Apel, and Martin Kuhlemann. A model of refactoring physically and virtually separated features. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, GPCE. ACM, 2009.

29 Christian Kästner, Paolo Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the Object-Oriented Programming Systems Languages and Applications*, OOPSLA. ACM, 2011.

30 Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A variability-aware module system. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA. ACM, 2012.

31 Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the International Conference on LISP and Functional Programming*, LFP. ACM, 1986.

32 Klaus H. Krippendorff. *Content Analysis: An Introduction to Its Methodology.* SAGE Publications, 2014.

33 Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *Proceedings of the International Conference on Software Engineering*, ICSE. IEEE, 1994.

34 Aditya Kumar, Andrew Sutton, and Bjarne Stroustrup. Rejuvenating C++ programs through demacrofication. In *Proceedings of the International Conference on Software Maintenance*, ICSM. IEEE, 2012.

35 Aditya Kumar, Andrew Sutton, and Bjarne Stroustrup. Rejuvenating C++ programs through demacrofication. In *Proceedings of the International Conference on Software Maintenance*, ICSM. IEEE, 2012.

36 Steinar Kvale. *InterViews: An Introduction to Qualitative Research Interviewing.* SAGE Publications, 1996.

37 David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the USENIX Security Symposium.* USENIX Association, 2001.

38 Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of International Conference on Software Engineering*, ICSE. ACM, 2010.

39 Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer. Morpheus: Variability-aware refactoring in the wild. In *Proceedings of the International Conference on Software Engineering*, ICSE. ACM, 2015.

40 Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, AOSD. ACM, 2011.

41 Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, FSE. ACM, 2013.

42 Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *Proceedings of the European Conference on Computer Systems*, EuroSys. ACM, 2006.

43 Bill McCloskey and Eric Brewer. Astec: A new approach to refactoring C. In *Proceedings of the European Software Engineering Conference and International Symposium on Foundations of Software Engineering*, ESEC/FSE. ACM, 2005.

44 Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. Investigating Preprocessor-Based Syntax Errors. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences*, GPCE. ACM, 2013.

45 Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, and Baldoino Fonseca. A catalogue of refactorings to remove incomplete annotations. *Journal of Universal Computer Science*, 2014.

46 Christopher A. Mennie and Charles L. A. Clarke. Giving meaning to macros. In *Proceedings of the International Conference on Program Comprehension*, ICPC. IEEE, 2004.

47 Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. Cowboys, ankle sprains, and keepers of quality: How is video game development different from software

development? In *Proceedings of the International Conference on Software Engineering*, ICSE. ACM, 2014.

48 Sarah Nadi and Ric Holt. The Linux kernel: A case study of build system variability. *Journal of Software: Evolution and Process*, 26(8), 2014.

49 Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computer Surveys*, 43(2), 2011.

50 Jeffrey L. Overbey, Farnaz Behrang, and Munawar Hafiz. A foundation for refactoring C with macros. In *Proceeding of the International Symposium on the Foundations of Software Engineering*, FSE. ACM, 2014.

51 Yoann Padioleau. Parsing C/C++ code without pre-processing. In *Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*. Springer, 2009.

52 T. Troy Pearse and Paul W. Oman. Experiences developing and maintaining software in a multi-platform env. In *Proceedings of the International Conference on Software Maintenance*, ICSM. IEEE, 1997.

53 Márcio Ribeiro, Paulo Borba, and Christian Kästner. Feature maintenance with emergent interfaces. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2014.

54 Márcio Ribeiro, Felipe Queiroz, Paulo Borba, Társis Tolêdo, Claus Brabrand, and Sérgio Soares. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, GPCE. ACM, 2011.

55 Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. Does the discipline of preprocessor annotations matter?: A controlled experiment. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences*, GPCE, 2013.

56 Robert C. Seacord. *The: 98 Rules for Developing Safe, Reliable, and Secure Systems*. Addison-Wesley, 2014.

57 Nieraj Singh, Celina Gibbs, and Yvonne Coady. C-CLR: A tool for navigating highly configurable system software. In *Proceedings of the AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, ACP4IS. ACM, 2007.

58 Henry Spencer and Geoff Collyer. `#ifdef` considered harmful, or portability experience with C news. In *USENIX Annual Technical Conference*, 1992.

59 Diomidis Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering*, 29(11), 2003.

60 Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Static analysis of variability in system software: The 90,000 `#ifdefs` issue. In *USENIX Annual Technical Conference*, 2014.

61 Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem. In *Proceedings of the Conference on Computer systems*, EuroSys. ACM, 2011.

62 Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer*, 14(5), 2012.

63 Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Dead or alive: finding zombie features in the linux kernel. In *Proceedings of the International Workshop on Feature-Oriented Software Development*, FOSD, 2009.

64 Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 47(1), 2014.

65 Federico Tomassetti and Daniel Ratiu. Extracting variability from C and lifting it to mbeddr. In *Proceedings of the International Workshop on Reverse Variability Engineering*, REVE, 2013.

66 Salvador Trujillo, Don Batory, and Oscar Diaz. Feature refactoring a multi-representation program into a product line. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, GPCE. ACM, 2006.

67 Marian Vittek. Refactoring browser with preprocessor. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, CSMR. IEEE, 2003.

68 Daniel Weise and Roger Crew. Programmable syntax macros. In *Proceedings of Interntinoal Conference on Programming Language Design and Implementation*, PLDI. ACM, 1993.

The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript

Michael Pradel[1] **and Koushik Sen**[2]

1 **TU Darmstadt**
Department of Computer Science
Germany
michael@binaervarianz.de

2 **University of California, Berkeley**
EECS Department
USA
ksen@berkeley.edu

Abstract

Most popular programming languages support situations where a value of one type is converted into a value of another type without any explicit cast. Such implicit type conversions, or type coercions, are a highly controversial language feature. Proponents argue that type coercions enable writing concise code. Opponents argue that type coercions are error-prone and that they reduce the understandability of programs. This paper studies the use of type coercions in JavaScript, a language notorious for its widespread use of coercions. We dynamically analyze hundreds of programs, including real-world web applications and popular benchmark programs. We find that coercions are widely used (in 80.42% of all function executions) and that most coercions are likely to be harmless (98.85%). Furthermore, we identify a set of rarely occurring and potentially harmful coercions that safer subsets of JavaScript or future language designs may want to disallow. Our results suggest that type coercions are significantly less evil than commonly assumed and that analyses targeted at real-world JavaScript programs must consider coercions.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, F.3.2 Semantics of Programming Languages, D.2.8 Metrics

Keywords and phrases Types, Type coercions, JavaScript, Dynamically typed languages

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.519

1 Introduction

In most popular programming languages, a value has a type that determines how to interpret the data represented by the value. To change the way a value is interpreted, programmers can convert a value of one type into a value of another type, called *type conversion*. In addition to explicit type conversions, or casts, many languages perform implicit type conversions, called *type coercions*. Type coercions are applied by the compiler, e.g., to transform an otherwise type-incorrect program into a type-correct program, or by the runtime system, e.g., to evaluate expressions that involve operands of multiple types. Both statically typed and dynamically typed languages use type coercions. For example, C and Python coerce numeric values from integer types to floating point types, and vice versa, and Java coerces instances of a subtype into an instance of a supertype. Some dynamically typed languages, such as JavaScript, make even wider use of type coercions.

© Michael Pradel and Koushik Sen;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 519–541

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Type coercions are a highly controversial language feature. On the one hand, they enable programmers to write concise code. On the other hand, type coercions can obfuscate a program, because the source code does not clarify which types of values a piece of code operates on, and even hide errors, because an unintended operation or an unintended loss of information may remain unnoticed.

This paper focuses on JavaScript, a language notorious for its heavy use of type coercions. Many operations that are considered illegal in languages with a stricter type system are legal in JavaScript. The reason is that JavaScript avoids throwing runtime type mismatch exceptions as much as possible so that programs can tolerate errors as much as possible. Although the rules for coercing types are well defined [10], even expert JavaScript developers struggle to fully comprehend the behavior of some code. For illustration, consider the following rather surprising examples:

- `"false" == false` evaluates to `false`, but `"0" == "false"` evaluates to `true`.
- `new String("a") == "a"` and `"a" == new String("a")` evaluate to `true`, but `new String("a") == new String("a")` evaluates to `false`.
- `[] << "2"` evaluates to `0`, `[1] << "2"` evaluates to `4`, and `[1,2] << "2"` evaluates to `0`.

Because type coercions can lead to such surprising behavior in JavaScript, it is common to assume that coercions are error-prone and therefore rarely used in practice. Based on these assumptions, existing static type inference and checking approaches for dynamically typed languages do not support type coercions at all [25, 2] or allow only a limited set of coercions [11, 4]. For example, Chugh et al. prohibit implicit coercions since they "often lead to subtle programming errors" [2].

While such assumptions about coercions are common, little is currently known about how type coercions are used in practice. This paper addresses this problem and asks the following research questions:

- RQ 1: How prevalent are type coercions in JavaScript, and what are they used for?
- RQ 2: Are type coercions in JavaScript error-prone?
- RQ 3: Do type coercions in JavaScript harm code understandability?

Answering these questions provides insights that enable informed decision making in at least three situations. First, developers of static and dynamic analyses benefit from the answers when deciding about how to handle coercions. Second, developers of safer subsets of JavaScript, such as *strict mode* [10], can use the answers to estimate the number of changes that a particular subset of JavaScript imposes on existing programs. Third, designers of future languages can benefit from the answers to decide whether and how to include type coercions.

To address these questions, this paper presents an empirical study of type coercions in real-world JavaScript web applications and in popular benchmark suites. As a result of the study, we can substantiate and refute several pieces of conventional wisdom, respectively. Our results include the following findings:

- Type coercions are widely used: 80.42% of all function executions perform at least one coercion and 17.74% of all operations that may apply a coercion do apply a coercion, on average over all programs. (RQ 1)
- In contrast to coercions, explicit type conversions are significantly less prevalent. For each explicit type conversion that occurs at runtime, there are 269 coercions. (RQ 1)
- We classify coercions into harmless and potentially harmful coercions, and we find that 98.85% of all coercions are harmless and likely to not introduce any misbehavior. (RQ 1)
- A small but non-negligible percentage (1.15%) of all type coercions are potentially harmful. Future language designs or restricted versions of JavaScript may want to forbid them.

For today's JavaScript programs, a runtime analysis can report these potentially harmful coercions. (RQ 2)
- Out of 30 manually inspected potentially harmful code locations, 22 are, to the best of our knowledge, correct, and only one is a clear bug. These results suggest that the overall percentage of erroneous coercions is very small. (RQ 2)
- Most code locations with coercions are monomorphic (86.13%), i.e., they always convert the same type into the same other type, suggesting that these locations could be refactored into explicit type conversions for improved code understandability. (RQ 3)
- Most polymorphic code locations (93.79%), i.e., locations that convert multiple different types, are conditionals where either some defined value or the `undefined` value is coerced to a boolean. (RQ 3)
- JavaScript's strict and non-strict equality checks are mostly used interchangeably, suggesting that refactoring non-strict equality checks into strict equality checks can significantly improve code understandability. (RQ 3)

All data gathered for this study and the full details of our results are available for download:

`http://mp.binaervarianz.de/ecoop2015`

In summary, this paper contributes the following:
- The first in-depth study of type coercions in real-world JavaScript programs.
- Empirical evidence that contradicts the common assumption that type coercions are rarely used and error-prone. Instead, we find that coercions are highly prevalent and mostly harmless.
- A classification of coercions into harmless and potentially harmful that may guide the development of safer subsets of JavaScript and future language designs.

2 Methodology and Subject Programs

To answer the research questions from the introduction, we perform a study of type coercions in real-world JavaScript programs. We dynamically analyze an execution of each program to record all runtime events that may cause a coercion (Section 2.1). Based on these data, we perform a set of offline analyses that summarize the runtime data into representations that allow for answering the research questions.

2.1 Dynamic Analysis

To gather runtime information about coercions, we instrument the program to intercept all runtime events that may cause a coercion or an explicit type conversion:
- *Unary and binary operations.* For unary and binary operations, the analysis records the source code location, the operator, the types of the input(s) and the output of the operation, and an abstraction (defined below) of the runtime values of the input(s) and the output of the operation.
- *Conditionals.* For each conditional evaluated during the execution, the analysis records the code location, the type and the abstracted runtime value of the evaluated expression, and the boolean value into which the expression gets coerced.
- *Function calls.* For each function call, the analysis checks whether the called function is any of the built-in functions that explicitly convert values from one type to another, i.e., `Boolean`, `Number`, and `String`. In this case, the analysis records the code location, as well as the types and the abstracted values of the function argument and the return value.

We abstract runtime values as follows: For booleans, `undefined`, `null`, and `NaN` (not a number), the analysis stores the value. For other numbers, it stores whether the value is zero or non-zero. For strings, it stores whether the string is empty or non-empty. For objects, including arrays, it stores whether the object has any own properties, i.e., whether it is an empty object or array. Furthermore, the analysis records whether an object has `valueOf` and `toString` methods that differ from the default implementation inherited from `Object` and, if an object has such a `valueOf` method, the type of `valueOf`'s return value. This extra type information allows us to identify values that explicitly define how to coerce them into another type.

Our implementation builds upon Jalangi [22], which instruments the JavaScript source code so that the analysis can intercept all necessary runtime events. For programs that execute on Node.js, we instrument the program on the file system. For web applications, we modify the Firefox browser so that it intercepts and instruments all JavaScript code before executing the code. The dynamic analysis creates a file that summarizes all recorded information for a program, and we analyze this file offline, i.e., after the execution.

For the purpose of this study, dynamic analysis has several benefits over a comparable static analysis. First, statically determining whether a code location coerces a type is impossible in general due to the highly dynamic nature of JavaScript. We believe that statically overapproximating potential coercions may skew the study results. Second, dynamic analysis enables us to reason about concrete runtime values, which is important for the qualitative part of our study, where we manually inspect coercions to determine whether coercions are harmful. Finally, dynamic analysis enables us to quantify how often coercions occur at runtime. On the downside, dynamic analysis misses coercions on paths that are not executed. We discuss implications of this limitation in Section 5.

2.2 Subject Programs

Our study considers three kinds of programs: the SunSpider benchmarks, the Octane benchmarks, and the top 100 most popular web sites according to Alexa[1]. For each benchmark, we analyze an execution of the benchmark in its default setup. For each web site, we load the start page and analyze all JavaScript code that gets executed, including code loaded from third-party libraries. In total, the study considers 138,979,028 runtime events from 132 programs. These events are generated by 321,711 unique source code locations.

In addition to the main research questions of this paper, this setup also allows us to address the question how accurately the benchmark suites, which are widely used to evaluate JavaScript engines, represent the type coercion-related behavior of real-world web applications.

3 Classification of Type Coercions

In this section, we propose a classification of all type coercions that may occur in JavaScript into likely harmless and potentially harmful coercions. This classification may serve three purposes. First, we use it to approximate the harmfulness of coercions in practice by classifying the coercions we observe in real-world programs. Second, the classification may provide a basis for defining a safer subset of JavaScript that forbids or warns about potentially harmful coercions. Such a subset may be enforced through runtime checks, similar to JavaScript's strict mode [10]. Third, the classification may guide future language designs that want to

---

[1] `http://www.alexa.com/topsites`, accessed on July 16, 2014

allow harmless coercions for conciseness but disallow potentially harmful coercions. The results of our study will help approaches of the second and third direction by providing empirical data about how often different kinds of coercions occur, i.e., how much code one would break by disallowing them.

Since developers may purposefully exploit the behavior of any type coercion, there is no clear-cut definition of when a coercion constitutes an error. The proposed classification is based on our own experience with JavaScript, on reports of the experience of others, e.g., in web forums, and on a comparison with other programming languages. We classify a coercion as *potentially harmful* if its semantics deviates from what is common in other, more strongly typed languages, such as C, Java, Python, or Ruby, if the operation that triggers the coercion has no intuitive meaning, or if the rules that determine which coercion to apply are very complex. We classify all other type coercions as harmless. The remainder of this section presents and illustrates our classification, which is summarized in Table 1.

3.1 Terminology

JavaScript has six basic types: The three primitive types boolean, number, and string, the special, single-value types `undefined` and `null`, and the object type, which includes arrays and functions. To simplify our classification, we use the following additional terms.

- A *quasi-number* is a primitive number or an object that defines a `valueOf` method that returns a primitive number. Developers can use `valueOf` to specify how to coerce an object, and the execution environment calls the method whenever the language requires a coercion.
- A *quasi-string* is a primitive string or an object that defines a `valueOf` method that returns a primitive string.
- A *wrapped primitive* is an object created with one of the built-in wrappers, `new Boolean()`, `new Number()`, and `new String()`.
- An *empty object* is an object without any own properties, e.g., the result of evaluating the object literal expression `{}`.
- An *empty array* is an array with length zero, e.g., the result of evaluating the array literal expression `[]`.
- A *defined value* is every value except for `undefined` and `null`.

3.2 Conditional-related Coercions

In JavaScript, values of all types may be used as conditions. Furthermore, all types may occur as the operand of the logical negation operator `!` and as the operands of the binary logical operators `&&` and `||`. The semantics are straightforward: All objects, all strings except the empty string, and all numbers except zero and `NaN` coerce to `true`. Note that all objects include objects whose `valueOf` returns `false`, empty arrays, and empty objects. All other values, including `undefined` and `null`, coerce to `false`. Because of these rather simple semantics and because using arbitrary types in conditionals is very common in JavaScript, we consider these coercions as harmless.

As an exception to the above classification, we classify wrapped primitives used as conditions or as operands of `!`, `&&`, and `||` as potentially harmful. Because all objects, including wrapped primitives, coerce to `true`, the semantics of wrapped primitives in conditionals differs from their primitive counterparts, which may surprise developers. For example, the wrapped primitive `new Boolean(false)` coerces to `true`. Popular guidelines [3] suggest to avoid wrapped primitives altogether.

Table 1 Classification of type coercions into likely harmless (✔) and potentially harmful (✖).

Operation	Type of operands	Example	Comment	Class.
Conditional:				
-	Wrapped primitives	`if (new Boolean(false))`	Behavior differs from primitives	✖
-	All other types	`if (someNumber)`	Coerced to boolean	✔
Unary operations:				
`+`, `-`, `~`	All except quasi-numbers	`-"abc"`	Coerced to number	✖
`!`	Wrapped primitives	`!(new Boolean(false))`	Behavior differs from primitives	✖
`!`	All except wrapped primitives	`!someNumber`	Coerced to boolean	✔
Binary operations:				
`-`, `*`, `/`, `%`	All except quasi-numbers	`"abc" * false`	Meaningful only for numbers	✖
`<<`, `>>`, `>>>`	All except quasi-numbers	`{} << 23`	Meaningful only for numbers	✖
`+`	Quasi-string and `undefined`, or quasi-string and `null`	`var x; x+="abc"`	Result contains `"undefined"` or `"null"`	✖
`+`	Quasi-string and a defined value	`"Names: " + arrayOfNames`	Common to construct strings	✔
`+`	Two non-quasi-strings	`false + [2,3]`	Confusing semantics	✖
`<`, `>`, `<=`, `>=`	All except two quasi-numbers and two quasi-strings	`[1,2] < function f() {}`	Meaningful only for numbers and strings	✖
`==`, `!=`	All types, unless both types are the same, or one is `undefined` or `null`	`0 == "false"`	Confusing semantics	✖
`==`, `!=`	One value is `undefined` or `null`	`someObject != null`	Common in conditionals	✔
`|`, `||`	Left type is `undefined`, right value is `0`	`x = (x | 0) + 1`	Common pattern to initialize counters	✔
`&`, `|`, `^`	All types, unless both are quasi-numbers, or counter initialization pattern from above	`[1,2] & "abc"`	Meaningful only for numbers	✖
`&&`, `||`	At least one wrapped primitive	`new Number(0) || new Boolean(false)`	Behavior differs from primitives	✖
`&&`, `||`	All except wrapped primitives	`someNumber && someBoolean`	Common in conditionals	✔

3.3 The Plus Operator

The semantics of the `+` operator is defined for numbers, where it means addition, and for strings, where it means concatenation. If any value that is neither a number nor a string is given to `+`, JavaScript coerces the value either into a primitive number or into a primitive string. Then, it applies string concatenation if at least one operand is a string and addition otherwise. The rules for deciding whether a value gets coerced into a number or a string are somewhat intricate. For most but not all values, JavaScript attempts to coerce the value into a number by calling `valueOf` and falls back on coercing into a string by calling `toString`. We refer to [10] for a full description of the rules.

Given that `+` has easy to understand semantics when being applied to two quasi-numbers or to two quasi-strings, we classify coercions that happen in these cases as harmless. We also classify coercions as harmless if they result from combining a quasi-string with any defined value because developers commonly use this pattern to concatenate strings. For example, `"Names:  " + arrayOfNames` causes a harmless coercion of an array into a string representation of the array. In contrast to these harmless coercions, we classify all other coercions triggered by `+` as potentially harmful because their semantics differ from more strongly typed languages. For example, the statements `var x; x+="abc";`, which yield the string `"undefinedabc"` instead of the probably expected `"abc"`, and the expression `false+{}`, which yields "false[object Object]", are classified as potentially harmful.

3.4 Arithmetic and Bitwise Operators

The unary `+` and `-` operators coerce their operand to a number. The arithmetic operators `-`, `*`, `/`, `%`, and the bitwise shift operators `<<`, `>>`, and `>>>` are defined for numbers, and applying them to any types except for two numbers triggers a coercion to number. Similar to the arithmetic operators, the unary bitwise `~` operator and the binary bitwise operators `&`, `|`, and `^` are defined for 32-bit integers. Applying these operators to any other values leads to a coercion of the operands into 32-bit integers.

Because all these operations are meaningful only for numbers and values that coerce into a number, we consider them as harmless when being applied to quasi-numbers, and as potentially harmful otherwise. For example, we consider the following operations as potentially harmful:

- `-"abc"`, which results in `NaN`
- `[1,2] & "abc"`, which yields `0`
- `{} << 23`, which yields 0

The above classification considers coercing a string to a number in an arithmetic operation as potentially harmful. The reason is that one arithmetic operator, `+`, has a different semantics than the other arithmetic operators, which may easily confuse developers: `23 - "5"` is interpreted arithmetically and yields `18`, but `23 + "5"` is interpreted as string concatenation and yields `"235"`.

As an exception to the above classification, we consider a common code idiom for initializing counter variables, e.g., `x = (x | 0) + 1`. This idiom initializes `x` to zero when the code is executed for the first time, and it increments `x` otherwise. That is, at the first execution, the idiom coerces `undefined` to `0`. Because this idiom has clear semantics and is commonly used, we classify coercions caused by this pattern as harmless.

3.5 Relational Operators

The semantics of the relational operators `<`, `>`, `<=`, and `>=` is defined for numbers (numeric comparison) and for strings (lexicographic order). JavaScript coerces any pairs of values that contain a non-number or a non-string to either a pair of numbers or a pair of strings, and then applies the respective operation. The rules for such coercions are the following: At first, JavaScript coerces any non-primitive into a primitive, where a `valueOf` method that returns a number is preferred over a `toString` that returns a string. Then, JavaScript coerces any remaining non-numbers into numbers, unless both primitives are strings, in which case the lexicographic order is computed.

Because relational operators have an intuitive semantics for pairs of numbers and for pairs of strings, we classify coercions that occur when combining two quasi-numbers or two quasi-strings as harmless. In contrast, we classify all other coercions as potentially harmful, because relational operations have no meaningful semantics for other types. For example, we consider `[1,2] < function foo() {}`, which yields `true`, as potentially harmful.

3.6 Equality Operators

JavaScript provides two kinds of (in)equality operators: the strict operators `===` and `!==`, and the non-strict operators `==` and `!=`. The strict operators never apply any type coercions; instead, they consider any two values of different types as unequal. In contrast, their non-strict counterparts coerce operands to evaluate whether they may be considered equal despite having different types. Because the coercion rules for non-strict equality operations are rather complex and sometimes unintuitive (see [10] for full details), guidelines [3] recommend to avoid non-strict equality operations. Examples for the somewhat surprising behavior of non-strict equality operations include:

- `"" == 0` yields `true` but `"false" == 0` yields `false` because comparing a string and a number leads to coercing the string into a number, where `""` is coerced into `0` and `"false"` is coerced into `NaN`.
- `true == "true"` yields `false` but `true == "1"` yields `true` because comparing a boolean and any non-boolean leads to coercing the boolean into a number, which coerces `true` to `1`.
- Both `new Number(5) == 5` and `5 == new Number(5)` yield `true` but `new Number(5) == new Number(5)` yields `false` because non-strict equality is non-transitive.

Because of their rather confusing semantics, we consider type coercions caused by non-strict equality operations as potentially harmful, including all the examples given above. The only exception to this classification are comparisons of arbitrary values with `undefined` or `null`. Because `undefined` and `null` essentially mean the same in a non-strict comparison, such comparisons are commonly used to check for defined values. For example, we classify the following check as harmless: `someObject != null`.

4 Type Coercions in the Wild

4.1 RQ 1: Prevalence of Type Coercions

In the following, we address the question how prevalent type coercions are in real-world JavaScript programs.

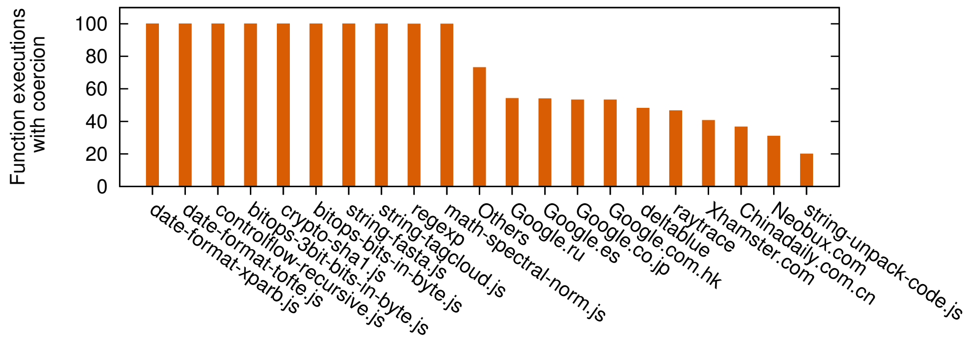


Figure 1 Percentage of function executions with at least one type coercion (top 10 and bottom 10 programs only).

4.1.1 Function Executions With At Least One Coercion

As a measure of the prevalence of type coercions, we assess during how many function executions at least one coercion occurs:

▶ **Definition 1** (Function executions with coercion, FEC)**.** The percentage FEC of function executions with a coercion is the number of function executions where at least one type coercion occurs between entering the function and exiting the function, excluding the execution of the function's callees, divided by the total number of function executions.

For example, executing the following program yields a FEC of 50% because `f()` does not perform any coercion, but calling `g()` triggers the coercion in line 6.

```
1  function f() {
2    g();
3    return 5 + 1;
4  }
5  function g() {
6    return 5 + true;
7  }
8  f();
```

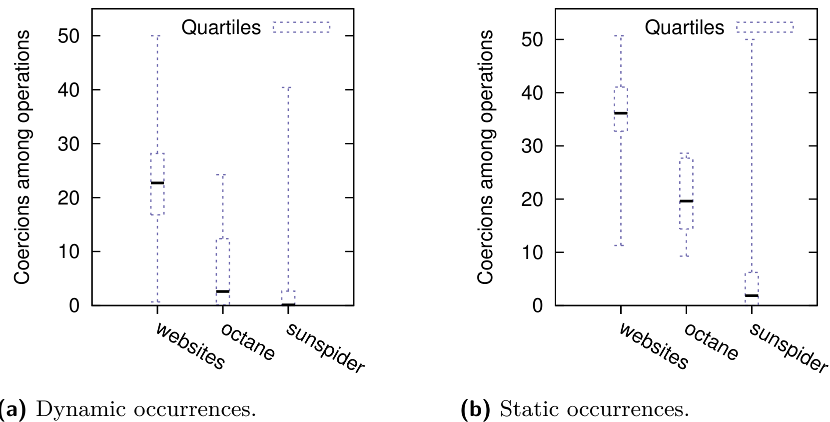
Over all programs we analyze, the FEC is 80.42%. This number indicates that coercions are a highly prevalent phenomenon that cannot be ignored when analyzing real-world JavaScript programs.

Figure 1 shows the FEC for a subset of all programs. The figure includes the ten programs with the highest percentage, the ten programs with the lowest percentage, and the average percentage of all other programs ("Others"). These results show that even programs with relatively few coercions still perform a non-negligible number of coercions. The figure excludes programs with less than 100 function executions because the FEC is not representative for these programs. In particular, the figure excludes several SunSpider benchmarks that perform less than 10 function executions and for which the FEC is 100%.

4.1.2 Coercions versus Non-coercions

As another measure of how prevalent coercions are, we compute how often an operation that could lead to a coercion does coerce one or more values:

▶ **Definition 2** (Coercions among operations, CAO)**.** The percentage CAO of coercions among operations is the number of operations that perform a coercion divided by the total number of executed unary and binary operations and evaluated conditionals that could perform a coercion.



(a) Dynamic occurrences.

(b) Static occurrences.

Figure 2 Prevalence of type coercions as percentage over all operations where type coercions may occur.

Figure 2 shows the *CAO* for the three groups of programs that we analyze. For each group, the figure shows the mean *CAO* over all programs from the group, the upper and lower quartiles, and the minimal and maximal *CAO*. We show two variants of the *CAO* measure. Figure 2a is based on the dynamic frequency of operations, i.e., each dynamic occurrence of an operation counts. In contrast, Figure 2b is based on static code locations, i.e., each static code location counts at most once.

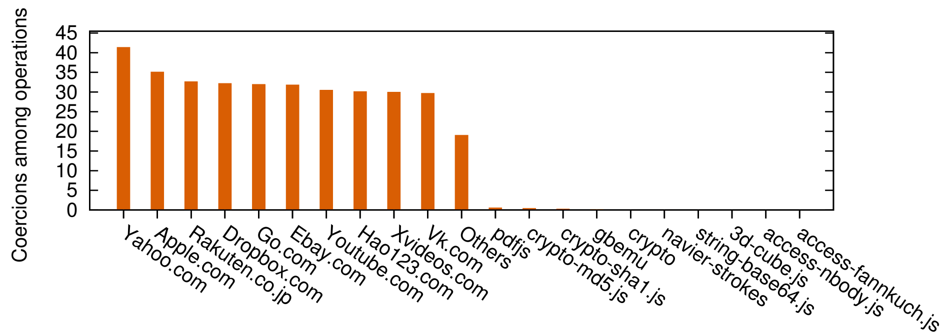
The results reveal two interesting properties. First, type coercions occur in a non-negligible fraction of all operations that may cause coercions. For web sites, 36.25% of all code locations that may coerce values indeed do it in the analyzed executions. Second, type coercions are significantly more prevalent in web sites than in the SunSpider and Octane benchmarks. These benchmark suites have been criticized to be unrepresentative for real-world JavaScript programs [21, 19], and our study confirms this observation for type coercions.

Figure 3 shows the *CAO* for the top 10 and bottom 10 programs. On many popular web sites, such as Yahoo.com and Apple.com, more than a third of all operations that could perform a coercion do perform a coercion. The 10 programs with the smallest *CAO* are all from the SunSpider or Octane benchmarks, which again shows that these benchmarks fail to accurately represent real-world web applications. The figure excludes programs with less than 100 observations because measuring their *CAO* does not provide representative results.

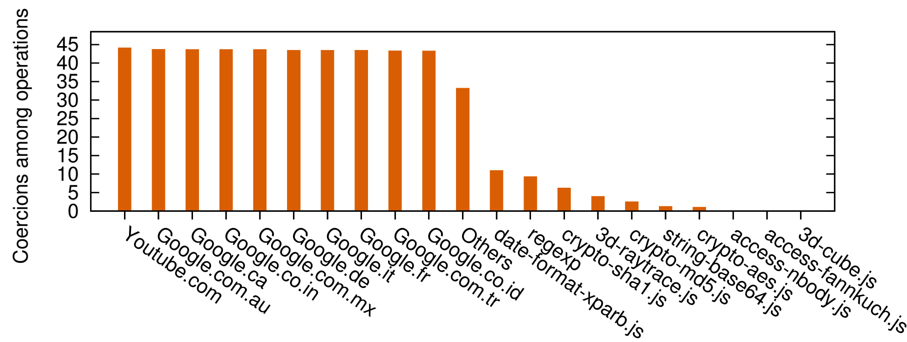
4.1.3 Kinds of Type Coercions

To better understand why coercions are so prevalent in real-world JavaScript programs, we analyze what kinds of coercions occur. Figure 4 shows the most and the least prevalent kinds of type coercions. The horizontal axis clusters similar kinds of coercions. For example, "number in conditional" means that a value of type `number` is coerced into a `boolean` because it occurs in a conditional, and "+-~ null" means that one of the arithmetic operators `+`, `-`, or `~` is applied to `null`, which leads to a coercion into the number zero.

The figure shows that conditionals and logical negations, which are typically used in conditionals, are the most prevalent kinds of coercion. Overall, coercions that result from conditionals or from operations that are typically used in conditionals (`!`, `&&`, and `||`) account for 93.01% of all coercions. This result suggests that analyses of JavaScript, such as type



(a) Dynamic occurrences.



(b) Static occurrences.

Figure 3 Programs with the highest and lowest prevalence of type coercions (measured as percentage over all operations where type coercions may occur).

inference and checking approaches, should at least consider these kinds of coercions because they occur frequently in practice.

4.1.4 Implicit versus Explicit Type Conversions

As an alternative to coercions, JavaScript developers can explicitly convert values from arbitrary types into booleans, numbers, and strings using the built-in functions `Boolean`, `Number`, and `String`. We measure how prevalent such explicit type conversions are and compare their prevalence to coercions. In total, we observe 20,407 explicit type conversions during the execution of all programs, and 5,497,545 coercions. That is, for every explicit type conversion that occurs, there are 269 implicit type conversions. We conclude that explicit conversions are used significantly less frequently than coercions. A possible explanation is that developers prefer the conciseness of coercions over the potentially increased code understandability through explicit conversions.

4.2 RQ 2: Harmfulness of Type Coercions

We address the question whether type coercions are error-prone in two ways. First, we measure how many type coercions are harmless and potentially harmful according to the classification from Section 3. Second, we manually inspect a sample of the potentially harmful type coercions to assess whether their behavior is intended or erroneous.

(a) Dynamic occurrences.

(b) Static occurrences.

Figure 4 Prevalence of different kinds of type coercions as percentage over all type coercions (top 10 and bottom 10 only). Horizontal axes: BIT, EQ, and REL mean bitwise, equality, and relational operations, respectively.

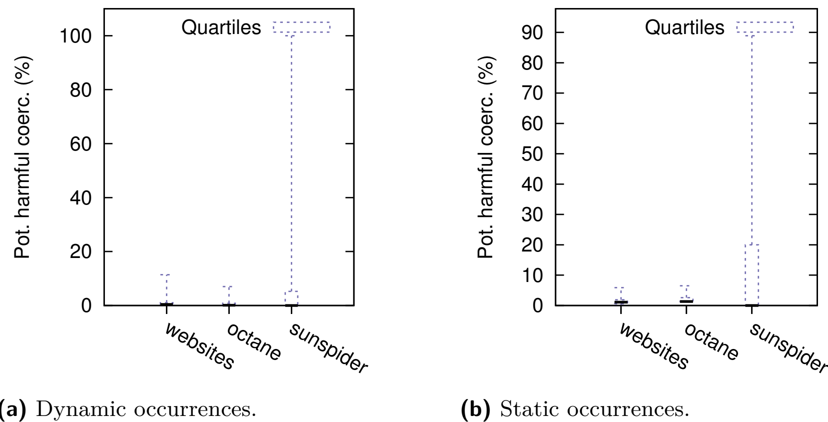
4.2.1 Harmless versus Potentially Harmful Coercions

Given the classification from Section 3, Figure 5 shows how many of all observed type coercions are potentially harmful. For most programs, a relatively small percentage of coercions is potentially harmful. For example, on average over all websites, only 1.25% of all dynamic occurrences of coercions are potentially harmful. The average over the coercions of all programs is 1.15%. Some of the SunSpider benchmarks are outliers in Figure 5, with up to 99.98% potentially harmful coercions. By manually inspecting these benchmarks we find that they indeed perform various potentially harmful coercions, which lead to a high percentage because these benchmark programs are relatively small.

Figure 6 shows the ten applications that have the highest and lowest percentage of potentially harmful coercions, respectively. Apart from the outliers in the SunSpider benchmarks discussed above, all programs have less than 12% potentially harmful coercions (for both dynamic and static occurrences).

4.2.2 Kinds of Potentially Harmful Coercions

Since the number of potentially harmful coercions is non-negligible, we further analyze what kinds of potentially harmful coercions occur. Figure 7 shows the 10 most prevalent and the 10 least prevalent kinds of potentially harmful coercions. The most prevalent potentially harmful coercion (by number of static occurrences) are non-strict (in)equality checks that compare



(a) Dynamic occurrences. **(b)** Static occurrences.

Figure 5 Percentage of potentially harmful coercions over all coercions.

two objects of different types, which is a common source of confusion. Several prevalent kinds of potentially harmful coercions involve `undefined`, such as concatenating `undefined` with a string, which yields a string that contains `"undefined"`, and relative operators applied to `undefined` and a number, which always yields `false`. We speculate that most of these coercions are caused by an `undefined` value that accidentally propagates through the program.

Our results suggest that developing analyses that warn programmers about potentially harmful coercions is a promising line of future work, e.g., along the lines of [11]. Our study provides empirical data on how many warnings such analyses may yield, so that developers of such analyses can focus on rarely occurring, potentially harmful coercions. Another direction for future work are techniques that prevent the `undefined` value from propagating in undesired ways.

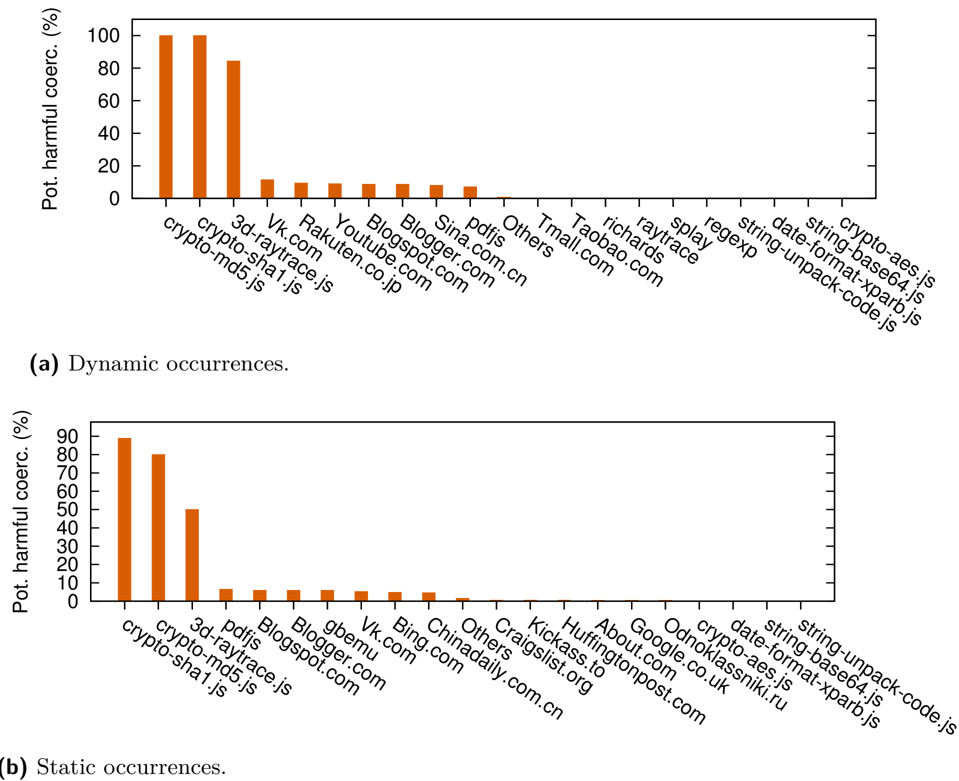
4.2.3 Binary Plus Operations

One of the most debated and potentially error-prone operators in JavaScript is the binary `+` operator. For example, a popular guideline [3] lists the operator as "problematic" and mentions that it is "a common source of bugs". To better understand how dangerous `+` is in practice, we analyze all occurrences of this operator.

Figure 8 shows what kinds of types `+` is applied on, including types that do not lead to any coercion. The labels on the horizontal axis also indicate whether a coercion occurs and if yes, whether we classify the coercion as potentially harmful. Most `+` operations apply to either two strings or to two numbers, i.e., they do not coerce any types. The most prevalent occurrence of `+` that coerces operands, number `+` string, is harmless. In total, only a very small percentage of all dynamic occurrences of `+` lead to a potentially harmful coercion.

We conclude from these results that the `+` operator is less dangerous than commonly expected. Programmers are disciplined enough to apply `+` (mostly) in situations where the operation does not cause any type coercion or where it applies a harmless coercion that has obvious semantics. That said, reconsidering the semantics of `+` in future language designs to reduce its complexity seems to be a good idea. To deal with today's JavaScript, checking for the rarely occurring potentially harmful usages of `+` is a promising endeavor for static or dynamic analyses.

4.2.4 Manual Inspection of Potentially Harmful Coercions



(a) Dynamic occurrences.

(b) Static occurrences.

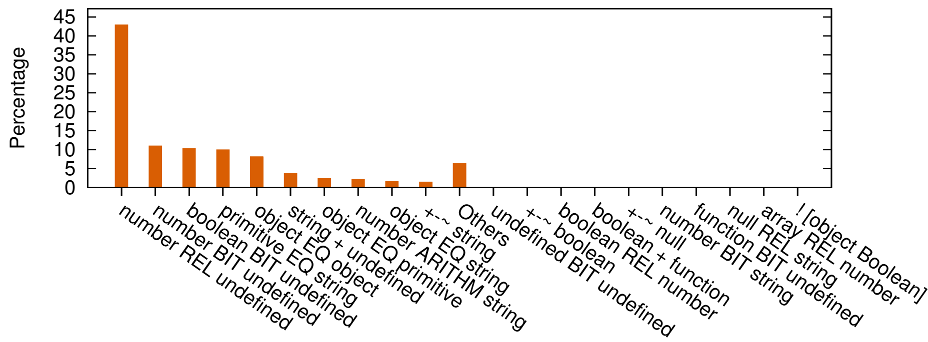
Figure 6 Programs with the highest and lowest percentage of potentially harmful coercions.

To gain further insights into the harmfulness of coercions, we manually inspect a random sample of all potentially harmful coercions. Out of the total of 1,329 unique code locations that perform at least one potentially harmful coercion, we inspect a random sample of size 30. Inspecting these coercions is non-trivial because most web sites obfuscate and minify their JavaScript code, and because we are not familiar with the implementation of the studied programs. Therefore, we cannot provide a clear-cut classification of all inspected locations. To the best of our knowledge, out of the 30 inspected locations:

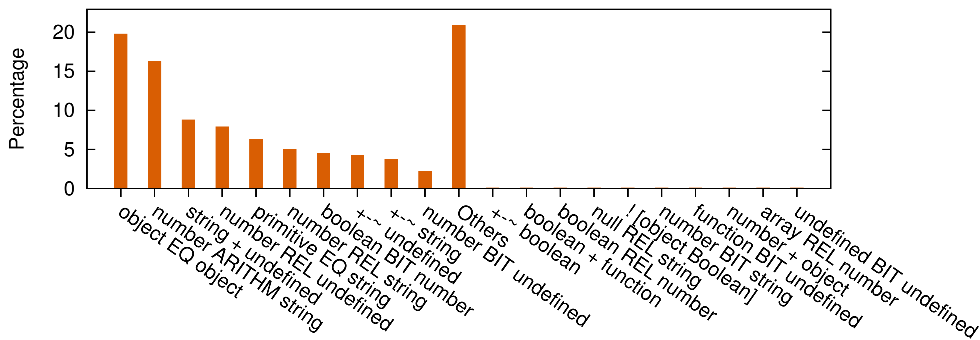
- 22 are probably correct,
- 1 is a clear bug,
- 3 may be buggy, but we cannot confirm any visible misbehavior, and
- 4 are unclear.

The clear bug was part of Sina.com.cn, a popular Chinese web portal. The page contained code that encodes various environment settings, such as the current timezone or the version number of the installed Flash player, into a single number. To access the environment settings, the page provides helper functions that are supposed to compare the number, `m`, against particular bit patterns:

```
1  timezone: function() {
2    return (m & 16384 == 16384) ? (new Date().getTimezoneOffset() / 60) : ""
3  },
4  flashVer: function() {
5    if (m & 8192 != 8192) {
6      return ""
7    }
8    ..
9  }
```



(a) Dynamic occurrences.



(b) Static occurrences.

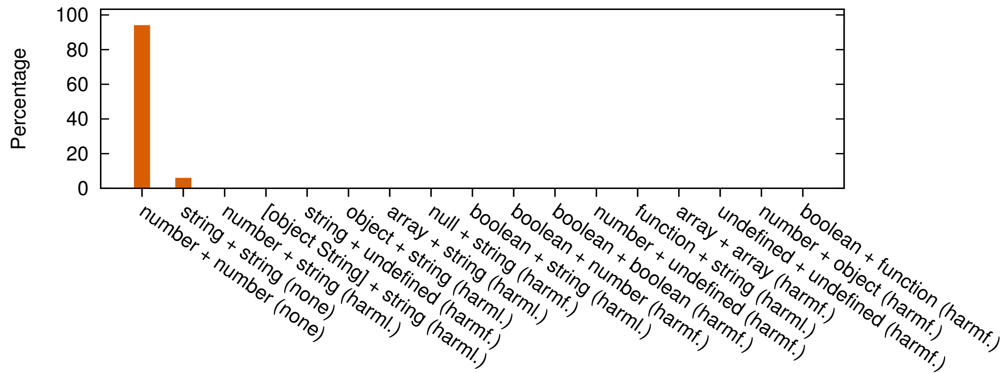
Figure 7 Kinds of potentially harmful type coercions (top 10 and bottom 10 kinds only). Horizontal axes: ARITHM, BIT, EQ, and REL mean arithmetic, bitwise, equality, and relational operations, respectively.

Unfortunately, these helper functions always return the same value because equality operators have precedence over the bitwise `&` operator. For example, in function `flashVer`, `8192 != 8192` always yields `false` and therefore `m & 8192 != 8192` always yields zero, which is coerced to `false`. This operation is classified as potentially harmful because it combines a number and `undefined` with the `&` operator. When we tried to reproduce the problem three days after gathering the data for this study, the corresponding code had been removed from the site.

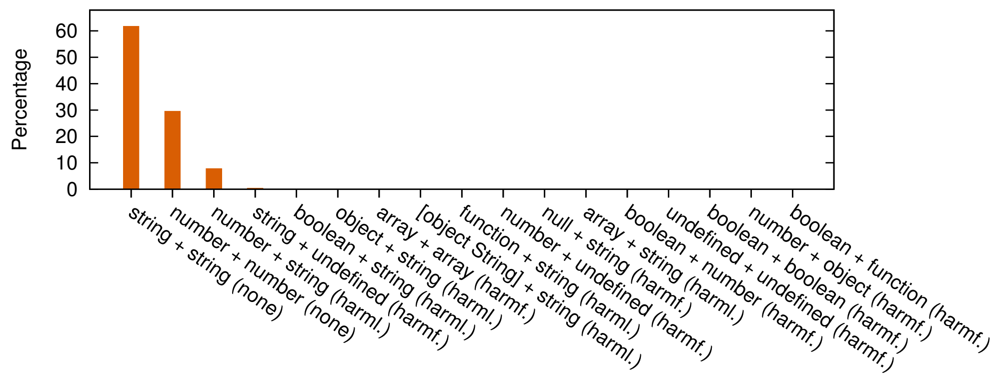
The three locations that we classify as maybe buggy include two locations that produce `NaN` because arithmetic operations are applied to `undefined`. The other location produces a string for debugging purposes but concatenates a string with `null` into `"Source-type: null"`, which may or may not help with debugging.

The 22 probably correct locations include several recurring patterns:

- Ten locations apply a relational or arithmetic operator to a number and a string, where the string coerces into a number. Representing numbers as strings is not recommended in general because `+` does not mean addition. However, the inspected code is correct because it does not use the `+` operator.
- Three locations concatenate a string to `undefined`, and then check whether the result matches a regular expression. This check could be implemented more efficiently, but it is correct.
- Two locations use non-strict equality checks even though they should only match if the operands have equal types. These locations are correct but would be easier to understand if strict equality was used instead.



(a) Dynamic occurrences.



(b) Static occurrences.

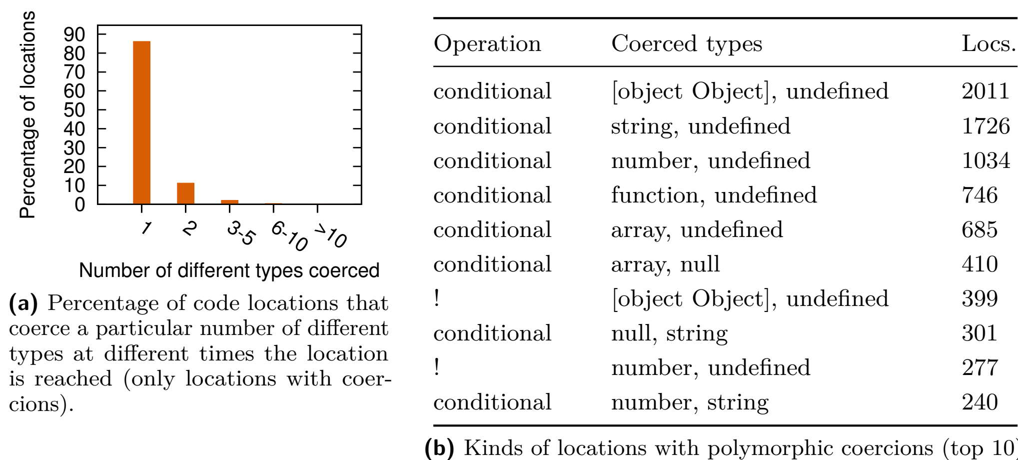
Figure 8 Kinds of plus operations.

Our results suggest that even among the potentially harmful coercions, most coercions do not cause incorrect behavior. Instead, some developers use JavaScript's coercion semantics in unusual yet correct ways.

4.2.5 Manual Inspection of Harmless Coercions

To validate the classification from Section 3, we also inspect a random sample of size 30 of all harmless coercions. We find that all inspected coercions are indeed harmless. As expected from Figure 4, most of the coercions (26 of 30) are related to conditionals. Moreover, we identify the following recurring patterns:

- Ten coercions are from conditionals that check if a value is defined before using it. Seven of them check if an object exists before accessing its properties, three check if a function exists before calling it.
- Three coercions check if an optional function argument is defined. JavaScript supports variadic functions and optional arguments are commonly used.
- Three coercions are instances of the initialization pattern discussed in Section 3.4.
- Four coercions are due to minified code that uses `!0` and `!1` as a concise way to express `true` and `false`, respectively.



(a) Percentage of code locations that coerce a particular number of different types at different times the location is reached (only locations with coercions).

Operation	Coerced types	Locs.
conditional	[object Object], undefined	2011
conditional	string, undefined	1726
conditional	number, undefined	1034
conditional	function, undefined	746
conditional	array, undefined	685
conditional	array, null	410
!	[object Object], undefined	399
conditional	null, string	301
!	number, undefined	277
conditional	number, string	240

(b) Kinds of locations with polymorphic coercions (top 10).

Figure 9 Polymorphism of code locations that perform coercions.

Overall, we conclude from these results that most coercions that occur in practice are harmless, which contradicts the common assumption that coercions are error-prone. We draw this conclusion for two reasons. First, even though our classification cannot rule out that some coercions classified as harmless cause errors, we believe that most JavaScript developers are aware of the semantics of the coercions that we classify as harmless. The results of manually inspecting coercions supports this assumption. Second, under the assumption that most of the analyzed programs perform the expected behavior, most of the coercions are likely to be correct, simply because coercions are very prevalent.

4.3 RQ 3: Influence on Understandability

To address the question whether and how type coercions influence the understandability of code, we analyze two particularly confusing coercion-related properties of code. First, we analyze the degree of polymorphism of code locations that apply coercions. Second, we present a detailed analysis of strict and non-strict equality checks, whose semantics often confuse developers and therefore may harm code understandability. Both analyses are proxy metrics that estimate to what degree coercions influence understandability. We leave a more detailed analysis of this question, e.g., through controlled experiments with developers [7], for future work.

4.3.1 Degree of Polymorphism

Do code locations with coercions always apply the same kind of coercion or do the types coerced at a particular location differ over time? Figure 9a shows the number of different types that are coerced at locations where we observed at least one type coercion. Most locations (86.13%) that apply a coercion always coerce values of the same types into each other, and very few locations (2.67%) apply three or more different kinds of coercions.

To understand why polymorphic code locations apply multiple kinds of coercions, Table 9b list the most prevalent kinds of polymorphic coercion locations and the types that they coerce. The table shows that the main reason for polymorphic coercions are conditionals that check whether some non-boolean value is defined. This pattern is common in JavaScript and a programmer that checks whether some value is defined will expect that the check may be applied to both a defined or an undefined value.

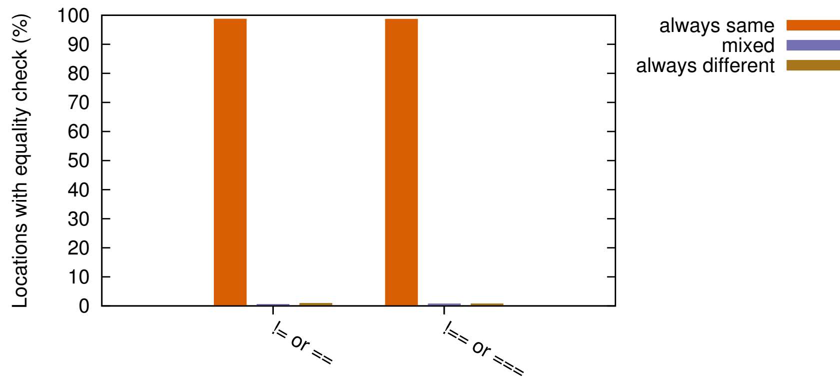


Figure 10 Percentage of code locations with (in)equality checks where all observed values have always the same type, always a different type, or both, respectively.

These results suggest that most coercions do not significantly harm code understandability, at least not because of polymorphic code locations. A more detailed study on how coercions influence human understanding of code, e.g., through a human study in the style of [7] remains for future work.

4.3.2 (In)equality Checks

A particularly intricate situation where JavaScript applies type coercions are (in)equality checks with `==` and `!=`. Guidelines [3] suggest to avoid `==` and `!=` altogether and to instead use their "non-evil" twins `===` and `!==`. Yet, we find that both kinds of equality checks are used in practice: In total, we observe 2,026,782 strict equality checks and 3,143,592 non-strict equality checks during the execution of all programs. To better understand to what degree coercions at equality checks influence code understandability, we measure how often a particular code location compares values of the same type (i.e., no coercion) and values of different types (i.e., coercion). "Same type" here means two types that can be trivially compared: (i) exactly the same type, (ii) in a `==` or `!=` check, any type compared to `undefined` or null, (iii) in a `===` or `!==` check, any type and `undefined`, or (iv) in a `===` or `!==` check, any non-primitive type and `undefined` or `null`.

Figure 10 gives the results for non-strict and strict comparisons. The figure shows that for almost all locations that compare values with `==` or `!=`, the values that occur at runtime are always of the same type. The results are very similar for comparisons with the strict operators `===` and `!==`.

We draw two conclusions from these results. First, developers seem to use non-strict and strict equality interchangeably. Under the assumptions that programmers fully understand the semantics of strict and non-strict equality and that they follow the advice to use strict equality when comparing values of the same type, one would expect that non-strict equality is mostly applied to different types. However, the results show that the percentage of locations that always compare the same types is almost equally high for non-strict and strict equality. Second, many non-strict equality checks could likely be refactored into strict checks. Since our study may not consider all paths, we cannot say with certainty that particular non-strict checks always compare values of the same type. Nevertheless, the results suggest that many code locations use `==` and `!=` without any need, and that these location could use `===` and `!==` instead.

5 Threats to Validity

The validity of the conclusions drawn from our results are subject to several threats. First, since our study is based on dynamic analysis, it ignores coercions triggered on paths not executed during the analysis. In particular, the results on polymorphic code locations (Figure 9a) potentially underestimate the number of coercions that occur at a location, and the results on equality checks (Figure 10) may classify a location as "always same" even though it may compare values of different types. Second, the classification of coercions into harmless and potentially harmful may be biased by the subjective experiences of the authors. We try to minimize this bias by considering informal experience reports of other JavaScript developers, e.g., in web forums and by comparing the behavior of JavaScript to other languages. Third, the subject programs of the study may not be representative for a larger population of programs. By focusing on the most popular web sites, code shared by multiple popular domains or popular third-party libraries may be overrepresented. Moreover, some of the benchmark programs contain generated JavaScript code, which may not be representative for human-written code. Fourth, the results of manually inspecting code that performs coercions are influenced by our limited ability to understand this code. To reduce this bias, we use a deobfuscation technique [18][2] to inspect minified and obfuscated code, and we interactively debug the inspected code locations. Finally, this study is limited to JavaScript, whose approach to type coercions occupies an extreme spot in the language spectrum. Our conclusions are for JavaScript and may not extend to other languages.

6 Related Work

Studying how programming languages are used in practice has a long history (for computer science standards), e.g., going back to a more than 40 years old study of Fortran programs by D. Knuth [12]. More recently, Richards et al. investigate the dynamic behavior of JavaScript programs and show that several dynamic features are widely used [21]. A study of Python programs draws similar conclusions and shows that the assumption that Python programmers only rarely use dynamic language features is false [9]. Another study [20] provides a detailed analysis of JavaScript's notorious `eval` function. In contrast to our work, none of these studies investigates type coercions. Nikiforakis et al. describe a large-scale study on how JavaScript-based web applications include code from third parties, and how these inclusions influence security [15]. Our work shares with [20] and [15] the idea to analyze in-depth how a particular language feature is used in the wild. Callau et al. describe a study of dynamic language features in Smalltalk [1]. In contrast to the above approaches and our work, they use static analysis. Their work focuses on reflection-related language features and finds that these features are used infrequently (in 1.76% of all methods).

There are various studies of how Java programmers use Java's language features. For example, Tempero et al. study the use of inheritance [24] and of fields [23]. Other work proposes an infrastructure for querying facts extracted from the source code of a corpus of Java programs, and shows how to use this infrastructure to answer various questions on how "typical" Java code is written [5]. Malayeri and Aldrich investigate whether Java programs could benefit from structural subtyping through a mixture of static and manual analysis [13]. Instead of analyzing how a feature could be used if it existed, we analyze how an already existing feature is used. To understand how much multiple dispatch is used and could be

[2] `http://jsnice.org`

used, Muschevici et al. study programs written in six languages that support this feature and in Java, respectively [14]. All these approaches are based on static analysis of the subject programs, whereas we use dynamic analysis.

Complementary to studying source code and its execution are studies on how human subjects react to particular languages or language features. Hanenberg presents such a study on whether a static type system reduces development time [7]. Our work raises several questions that could be addressed in similar studies, e.g., how type coercions influence program understandability, or whether the conciseness of code written with coercions outweighs the potential error-proneness of coercions during development. Ocariza et al. perform studies of JavaScript errors [17] and their root causes [16]. They do not identify coercions as a particular cause of errors, which matches our finding that most coercions do not cause misbehavior.

Several approaches for inferring and checking types in JavaScript programs have been proposed, some of which raise errors on type coercions. A type system for a subset of JavaScript by Thiemann [25] reports all coercions as errors, presumably under the assumption that coercions are generally erroneous. A statically typed dialect of JavaScript, called "Dependent JavaScript", prohibits type coercions because they "often lead to subtle programming errors" [2]. Our work contradicts this assumption based on empirical evidence showing that most coercions are harmless. A type analysis by Jensen et al. warns developers about particular kinds of coercions [11]. We also classify some of them as potentially harmful, e.g., coercing `undefined` to a number. Our results could help to reduce the number of warnings of their analysis by focusing on kinds of coercions that occur rarely and that are potentially harmful. Other type inference and checking approaches for JavaScript [8, 6] do not explicitly discuss if and how they handle coercions. Furr et al. propose a profile-guided static typing approach for Ruby [4]. They report that some coercions cause type errors that forced Furr et al. to refactor code to avoid coercions. Supporting type coercions in a static analyses is non-trivial and researchers need guidance on whether and how to support coercions. Our study provides empirical evidence that supporting type coercions is vital, along with guidance on which coercions to address first.

JavaScript's strict mode disallows some language features that are generally considered as dangerous. However, strict mode does not change the semantics of type coercions. A complementary approach to strict mode, called *restrict mode*[3], forbids several potentially harmful type coercions. At the time of this writing, the coercions allowed by restrict mode and the coercions that we classify as harmless overlap partly. For example, restrict mode does not warn about wrapped primitives in conditionals, whereas we classify them as potentially harmful, but it forbids concatenating strings and arrays, whereas we classify this operation as harmless. We believe that an empirical study of coercions provides a good base for an informed decision about which coercions to allow or disallow in a safer JavaScript subset.

7 Conclusion

This paper presents the first in-depth analysis of implicit type conversions in real-world JavaScript programs. In reference to the title of this paper, we show that most coercions are "good", few coercions are "bad", and some coercions are "ugly" but nevertheless correct. Based on the results of this study, we draw the following conclusions about real-world JavaScript programs and analyses that target them:

[3] `http://restrictmode.org`

- Type coercions are widely used and analyses that target realistic programs should take them into account.
- Most coercions are not erroneous, and even among those kinds of coercions that seem error-prone, most coercions do not cause any misbehavior.
- The previous two conclusions lead to the third one: Research on static analyses for JavaScript faces the challenge of checking programs with various harmless coercions.
- Most coercions occur in conditions and conditional-related operations, where some value is coerced into a boolean. Static analyses that address these coercions will handle a large part of all coercions.
- A very small subset of all coercions are potentially harmful because their semantics may surprise developers. Restricted variants of JavaScript and future language designs can prohibit these coercions while still supporting most existing code.
- Most code locations that coerce types are monomorphic, and most polymorphic locations are unsurprising because they are related to conditionals that check for undefined values. That is, polymorphism caused by coercions degrades code understandability only marginally.
- Developers use non-strict and strict equality almost interchangeably. Automated refactoring approaches that soundly transform non-strict into strict equality operations seem a promising direction for future work.
- We confirm earlier results showing that the SunSpider and Octane benchmarks do not accurately represent real-world web sites [21, 19], and we extend the scope of this finding to type coercions.

Given the increasing importance of JavaScript as a language for web, mobile, desktop, and server applications, we believe that our work is an important step toward understanding how developers use JavaScript and toward aligning future research activities with the real-world usage of the language.

Acknowledgments. This research is supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, by the German Research Foundation (DFG) within the Emmy Noether Project "ConcSys", by NSF Grants CCF-1423645 and CCF-1409872, by a gift from Mozilla, and by a Sloan Foundation Fellowship. Thanks to Liang Gong for numerous discussions on type coercions and JavaScript, and to the anonymous reviewers for their valuable feedback.

References

1 Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How (and why) developers use the dynamic features of programming languages: the case of smalltalk. *Empirical Software Engineering*, 18(6):1156–1194, 2013.

2 Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 587–606, 2012.

3 Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.

4 Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 283–300. ACM, 2009.

5 Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi-Reghizzi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An empirical investigation

into a large-scale Java open source code repository. In *Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2010.

6 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming (ESOP)*, pages 256–275, 2011.

7 Stefan Hanenberg. An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 22–35, 2010.

8 Phillip Heidegger and Peter Thiemann. Recency types for analyzing scripting languages. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 200–224, 2010.

9 Alex Holkner and James Harland. Evaluating the dynamic behaviour of Python applications. In *Australasian Computer Science Conference (ACSC)*, pages 17–25, 2009.

10 Ecma International. ECMAScript language specification, 5.1 edition, June 2011.

11 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Symposium on Static Analysis (SAS)*, pages 238–255. Springer, 2009.

12 Donald E. Knuth. An empirical study of FORTRAN programs. *Software Practice and Experience*, pages 105–133, 1971.

13 Donna Malayeri and Jonathan Aldrich. Is structural subtyping useful? An empirical study. In *European Symposium on Programming (ESOP)*, pages 95–111, 2009.

14 Radu Muschevici, Alex Potanin, Ewan D. Tempero, and James Noble. Multiple dispatch in practice. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 563–582, 2008.

15 Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *CCS*, pages 736–747, 2012.

16 Frolin S. Ocariza Jr., Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. An empirical study of client-side JavaScript bugs. In *Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 55–64, 2013.

17 Frolin S. Ocariza Jr., Karthik Pattabiraman, and Benjamin G. Zorn. JavaScript errors in the wild: An empirical study. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–109, 2011.

18 Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties from "big code". In *Principles of Programming Languages (POPL)*, pages 111–124, 2015.

19 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.

20 Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do - a large-scale study of the use of eval in JavaScript applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 52–78, 2011.

21 Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2010.

22 Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 488–498, 2013.

23 Ewan D. Tempero. How fields are used in Java: An empirical study. In *Australian Software Engineering Conference (ASWEC)*, pages 91–100, 2009.

24 Ewan D. Tempero, James Noble, and Hayden Melton. How do Java programs use inheritance? An empirical study of inheritance in Java software. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 667–691. Springer, 2008.

25 Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *European Symposium on Programming (ESOP)*, pages 408–422, 2005.

A Pattern Calculus for Rule Languages: Expressiveness, Compilation, and Mechanization

Avraham Shinnar, Jérôme Siméon, and Martin Hirzel

IBM Research, USA
{shinnar,simeon,hirzel}@us.ibm.com

Abstract

This paper introduces a core calculus for pattern-matching in production rule languages: the Calculus for Aggregating Matching Patterns (CAMP). CAMP is expressive enough to capture modern rule languages such as JRules, including extensions for aggregation. We show how CAMP can be compiled into a nested-relational algebra (NRA), with only minimal extension. This paves the way for applying relational techniques to running rules over large stores. Furthermore, we show that NRA can also be compiled back to CAMP, using named nested-relational calculus (NNRC) as an intermediate step. We mechanize proofs of correctness, program size preservation, and type preservation of the translations using modern theorem-proving techniques. A corollary of the type preservation is that polymorphic type inference for both CAMP and NRA is NP-complete. CAMP and its correspondence to NRA provide the foundations for efficient implementations of rules languages using databases technologies.

1998 ACM Subject Classification I.2.5 Programming Languages and Software: Expert systems, D.3.3 Language Constructs and Features: Patterns, H.2.3 Languages: Query Languages

Keywords and phrases Rules, Pattern Matching, Aggregation, Nested Queries, Mechanization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.542

Supplementary Material ECOOP Artifact Evaluation approved artifact available at
http://dx.doi.org/10.4230/DARTS.1.1.8

1 Introduction

Production rules are popular for business intelligence (BI) applications, because they can encode complex data-centric policies in a flexible manner [24]. Rules often appeal to business users, as they are easy to understand, extend, and modify. Languages for production rules go back to OPS5 [19], and modern specimens include JRules [24] and Drools [5]. These languages rely heavily on pattern matching and, more recently, on aggregation.

Figure 1 shows an example production rule with aggregation. It consists of a condition (`when`, Lines 2–5) and an action (`then`, Line 6). The condition binds variable `C` to a `Client` working memory element (WME), and aggregates all `Marketer` WMEs `M` for whom `C` belongs to `M`'s bag of clients. The aggregation uses the `collect` operator to create a bag, and binds that bag to `Ms`. Finally, Line 6 creates a new WME that materializes the mapping from `C` to `Ms`. This mapping could then be consulted in other rules, for instance: when a crucial client event happens, then notify all responsible marketers.

While production rule languages are starting to support aggregation, they do not yet have a good story for running aggregates on large and/or distributed data sets. Rules engines are usually centralized, tied to their internal data representation, and not readily applicable to other stores. Ideally, we would like to run production rules over distributed stores efficiently and leverage existing algorithms for large-scale data processing. To accomplish this, we

© Avraham Shinnar, Jérôme Siméon, and Martin Hirzel;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 542–567

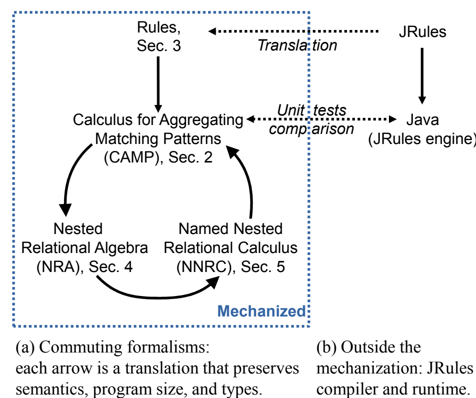
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
1  rule FindMarketers {
2    when {
3      C:  Client();
4      Ms: aggregate { M: Marketer(clients.contains(C.id)); }
5            do { collect {M}; }
6    } then { insert new C2Ms(C, Ms); }
7  }
```

Figure 1 In JRules, compute a reverse mapping from clients to marketers.

translate rules to a database algebra suitable for optimization and distributed execution. Because modern rules languages support complex data models and nesting (e.g., JRules uses an object-oriented data model and supports nested aggregation), we target a Nested Relational Algebra [14] (NRA). Optimization techniques for such algebras extend those already available in relational systems, and have been used to support efficient evaluation for a variety of nested data models, from OO [6, 14] to XML [27, 31].

This paper proposes the Calculus for Aggregating Matching Patterns (CAMP), a core calculus for production rule languages. The essence of patterns is that they have two implicit inputs: the data item that is the subject of the match, and an environment of variables previously bound by the match. The main novelty of CAMP is that it exposes the data flow inherent to the pattern matching of rules. Figure 2 gives an outline for the main languages covered in the paper and their relationships. Figure 2(b) shows the real JRules language which is the motivation for our work, which gets compiled and executed using a Rete-based rules engine [20]. This paper studies the properties of CAMP, proving that it has the same expressiveness as Nested Relational Algebra (NRA) and Named Nested Relational Calculus (NNRC), as depicted in Figure 2(a). The constructive proof includes a translation from CAMP to NRA, suitable as a first step towards an efficient compiler.



(a) Commuting formalisms: each arrow is a translation that preserves semantics, program size, and types.

(b) Outside the mechanization: JRules compiler and runtime.

Figure 2 CAMP formalization & JRules.

From a data processing perspective, the challenges in that translation include encoding the input datum and environment, and on the output side, encoding recoverable errors (caused by non-matching data) to be propagated. From a rules language perspective, the main challenge is to capture a representative yet minimal subset in CAMP. To make the connection between CAMP and productions rules clearer, we provide a Rules language with a syntax that parallels that of JRules and its corresponding encoding in CAMP. Our implementation includes a way to translate JRules into that formal Rules language and runs unit tests to check that the mechanized semantics yields the same results as the real JRules compiler. In that context, our approach is similar to prior work on the database-supported execution of programming languages with embedded queries, such as LINQ [11] compiled to a relational engine using Ferry [22]. One interesting outcome of our work is to show that one can offload the full pattern matching logic underlying production rules to a database engine. The main contributions of the paper are as follows:

- CAMP (Section 2), which captures the matching semantics for production rules with aggregates. The calculus is simple enough for formal reasoning and expressive enough to

support a Rules language (Section 3) that models large subsets of JRules [24].
- A full translation (Section 4) from CAMP to the NRA from [14] with only a minimal extension. This opens the door for taking advantage of the database literature for optimizing and distributing production rule matching over large stores.
- A reverse translation (Sections 5 and 6) from NRA back to CAMP, using the NNRC from [35] as scaffolding, showing that all three languages are equally expressive.
- Type systems for all three languages and proofs that the translations preserve types (Section 7). As a corollary, polymorphic type inference for CAMP is NP-complete.
- A mechanization of all the proofs of semantics preservation, program size, and type correctness using the Coq proof assistant [16] (see peer-reviewed artifact).

2 CAMP

The Calculus for Aggregating Matching Patterns (CAMP) models the query fragment of traditional production rules language such as OPS5 [19], JRules [24], or Drools [5]. The query fragment in these languages is a (possibly nested) pattern that is matched against working memory. To model this query style, CAMP patterns scrutinize an (implicit) datum that we call **it**. Additionally, to support naming matched fragments of **it**, CAMP patterns also refer to an (implicit) environment **env** that maps variables to data. CAMP expressions all return data, the result of the query. This is often a record containing variables bound by the pattern. Patterns can also fail if they do not match the given data. This failure is not fatal, and can trigger alternative pattern matching attempts.

2.1 Syntax

Definition 1 presents the syntax for CAMP. Section 2.3 presents the formal semantics; here we give an informal description.

▶ **Definition 1** (CAMP syntax).

$$\begin{aligned}\text{(patterns) } p ::= {} & d \mid \oplus p \mid p_1 \otimes p_2 \mid \mathbf{map}\ p \mid \mathbf{assert}\ p \mid p_1 \| p_2 \\ & \mid \mathbf{it} \mid \mathbf{let\ it} = p_1\ \mathbf{in}\ p_2 \mid \mathbf{env} \mid \mathbf{let\ env} \mathrel{+}= p_1\ \mathbf{in}\ p_2\end{aligned}$$

Going in order of presentation, d allows arbitrary (constant) data to be the result of a CAMP pattern. The full data model is presented in Section 2.2. The next two constructs allow unary ($\oplus$) and binary ($\otimes$) operators to process the result of a pattern or patterns. The set of operators is described in Section 2.2 and includes operations for constructing and manipulating records and bags, key components of our data model.

The **map** p construct maps a pattern p over the implicit data **it**. Assuming that **it** is a bag, the result is the bag of results obtained from matching p against each datum in **it**. Failing matches are skipped. The **assert** p construct allows a pattern p to conditionally cause match failure. If p evaluates to false, matching fails, otherwise, it returns the empty record $[\,]$. The $p_1 \| p_2$ construct allows for recovery from match failure: if p_1 matches successfully, p_2 is ignored; otherwise, if p_1 fails to match, p_2 is evaluated.

Finally, we come to the constructs that deal with the implicit datum being matched and the environment. The **it** construct obtains the datum that is being matched. The **let it** $= p_1$ **in** p_2 construct allows this implicit datum to be altered. Similarly, **env** reifies the current environment as a record. This reified environment can then be manipulated via standard record operators. The final construct, **let env** $\mathrel{+}= p_1$ **in** p_2, allows a pattern to add new bindings to the environment. The result of matching p_1 must be a record, which is

interpreted as a reified environment. If the current environment is compatible with the new one (all common attributes have equal values) then they are merged and the pattern p_2 is evaluated with the merged environment. If they are incompatible, the pattern fails. This mimics the standard convention in pattern matching languages that multiple bindings of a pattern variable must all bind to the same value.

As an example, the CAMP version of the JRules pattern `C: Client()` in Line 3 of Figure 1 is **let env** += **assert it**.*type* = "Client" **in let env** += $[C : \textbf{it}]$ **in env**. The pattern first asserts that **it** has type "Client". If **assert** succeeds, it returns the empty record, so the first **let env** leaves the environment unchanged. The next part of the pattern attempts to add $[C : \textbf{it}]$ to **env**. That succeeds if either C is not yet bound, or C is already bound to the same datum. The idiom **let env** += **assert** p_1 **in** p_2 is so common that we write $p_1 \wedge p_2$, for example, **it**.*type* = "Client" $\wedge$ **let env** += $[C : \textbf{it}]$ **in env**.

The example also relies on a fundamental data model and operators. For instance, **it**.*type* retrieves a record attribute, the = operator checks equality, and $[A : d]$ constructs a record. Those are shared across CAMP, NRA, and NNRC and are described next.

2.2 Data Model and Operators

Values in our data model, the set $\mathcal{D}$, are atoms, records, or bags. We assume a sufficiently large set of atoms $a, b, ...$ including numbers, strings, Booleans and a null value written *nil*. A bag is a multiset of values in $\mathcal{D}$; we write $\emptyset$ for the empty bag and $\{d_1, ..., d_n\}$ for the bag containing the values $d_1, ..., d_n$. A record is a mapping from a finite set of attributes to values in $\mathcal{D}$, where attribute names are drawn from a sufficiently large set $A, B, ...$ We write $[\,]$ for the empty record and $[\overline{A_i : d_i}]$ for the record mapping A_i to d_i.

Our data model uses bags instead of sets to naturally support aggregation. For instance, given employee records, a projection can obtain the bag of salaries, which can then be averaged. A set would yield the wrong result, as it would omit duplicate salaries. Other work uses bags for similar reasons [21]. Our data model supports arbitrary nesting of bags and records to model object-oriented rule languages such as JRules. This nesting increases expressiveness and simplifies the treatment of aggregation, group-by, and nested queries.

Records x and y are compatible if $\forall A \in \mathrm{dom}(x) \cap \mathrm{dom}(y), x(A) = y(A)$. We define the sum $x+y$ of compatible records as their union $x \cup y$, and leave it undefined for non-compatible records. An *operator* is a pure function defined on only its explicit operands; it does not access any implicit input (e.g., **it** or **env** in CAMP). Given operands of the correct types, operators are total. We factor out operators from each of our three languages (CAMP, NRA, and NNRC) to keep the languages small and focused on the essentials. Definition 2 presents a set of basic unary and binary operators on our data model.

▶ **Definition 2** (Operators).

$$
\begin{array}{llcl}
\text{(uops)} & \oplus d & ::= & \mathit{identity}\, d \mid \neg d \mid \{d\} \mid \#d \mid \mathit{flatten}\, d \mid [A\!:\!d] \mid d.A \mid d\!-\!A \\
\text{(bops)} & d_1 \otimes d_2 & ::= & d_1 = d_2 \mid d_1 \in d_2 \mid d_1 \cup d_2 \mid d_1 * d_2 \mid d_1 + d_2
\end{array}
$$

In order of presentation, the unary operators do the following:

$\mathit{identity}\, d$	returns d.
$\neg d$	negates a Boolean.
$\{d\}$	constructs a singleton bag containing the value d.
$\#d$	returns the number of elements in a bag.
$\mathit{flatten}\, d$	flattens a bag of bags.

$[A\!:\!d]$ constructs a record with a single attribute A containing the value d.
$d.A$ accesses the value associated with attribute A in record d.
$d-A$ returns a record with all attributes of d except A.

Each of the last three operators involves a datum d and a statically given attribute A. Once the attribute is fixed, they are unary operators on the datum. For instance, $d.type$ is the unary operator for retrieving attribute *type* of data d.

In order of presentation, the binary operators do the following:
$d_1 = d_2$ compares two data for equality.
$d_1 \in d_2$ returns true if and only if d_1 is an element of bag d_2.
$d_1 \cup d_2$ returns the union of two bags.
$d_1 * d_2$ concatenates two records, favoring d_1 if there are overlapping attributes.
$d_1 + d_2$ returns a singleton bag containing the concatenation of the two records if they are compatible, and returns $\emptyset$ otherwise.

In [35], the record concatenation operators are denoted $\times$ and $\bowtie$, but we use $*$ and $+$ instead to reserve $\times$ and $\bowtie$ for the NRA cross product and join on bags (see Section 4.1). Some of the operators, such as bag and record construction, are fundamental, since we need them to properly manipulate our data model. But since operators are simple functions, it is easy to add more. For instance, # counts elements of a bag, and we could easily add other unary reduction operators for summation or finding a minimum. Note that even record concatenation with $d_1 + d_2$ is total, returning the empty bag if the records are incompatible. We defer the discussion of type errors, which can be detected statically, to Section 7.

2.3 Semantics

The syntax and an informal description of CAMP were given in Section 2.1. Figure 3 presents a big-step operational semantics for CAMP. A mechanization of these semantics using the Coq proof assistant [16] is available in the companion artifact for this paper.

The relation $\sigma \vdash p \,@\, d \Downarrow_r d?$ relates an environment (record) σ, a pattern p, and a datum d (the implicit datum **it**) with an output $d?$. The subscript $_r$ on the relation stands for "rules", to distinguish the semantics from those of NRA and NNRC presented later in the paper. The output is a member of the lifted data domain $\mathcal{D}? = \mathcal{D} + \mathbf{err}$, representing either the returned datum or match failure.

The $\Downarrow_r$ relation is partial; malformed patterns and patterns that are given the wrong type of data may not admit any derivations. This is distinct from match failure, which is recoverable, and is internalized in the relation. Section 7 will present a type system for CAMP with a soundness result guaranteeing that well-typed patterns matching appropriately typed data can always derive a result (possibly **err**, indicating match failure).

The rules for constants and operators are standard. The rules for **map** conspire to evaluate the given pattern over each element of the datum. Pattern match failures are ignored and the rest of the results are gathered as the result. The Assert and OrElse rules allow patterns to explicitly cause and recover from an **err**. To enable easy sequencing with **let env**, **assert** true returns an empty record.

The data that is being matched is obtained via **it**, and temporarily replaced using **let it** $= p_1$ **in** p_2, where p_2 is evaluated with **it** set to the result of evaluating p_1.

The environment is obtained using **env**, which reifies it as a record. New bindings are added to the environment using **let env** $+= p_1$ **in** p_2. If p_1 evaluates to a record that is compatible (defined in Section 2.2) with the current environment, they are joined and used as

$$\frac{}{\sigma \vdash d_0 \,@\, d \Downarrow_r d_0}\text{(Constant)} \qquad \frac{\sigma \vdash p_1 \,@\, d \Downarrow_r d_1 \quad \sigma \vdash p_2 \,@\, d \Downarrow_r d_2 \quad d_1 \otimes d_2 = d_3}{\sigma \vdash p_1 \otimes p_2 \,@\, d \Downarrow_r d_3}\text{(Binary Op)}$$

$$\frac{\sigma \vdash p \,@\, d \Downarrow_r d_0 \quad \oplus d_0 = d_1}{\sigma \vdash \oplus p \,@\, d \Downarrow_r d_1}\text{(Unary Op)} \qquad \frac{\sigma \vdash p \,@\, d \Downarrow_r \mathbf{err} \quad \sigma \vdash \mathbf{map}\ p \,@\, s \Downarrow_r s_0}{\sigma \vdash \mathbf{map}\ p \,@\, \{d\} \cup s \Downarrow_r s_0}\text{(Map }\mathbf{err}\text{)}$$

$$\frac{}{\sigma \vdash \mathbf{map}\ p \,@\, \emptyset \Downarrow_r \emptyset}\text{(Map }\emptyset\text{)} \qquad \frac{\sigma \vdash p \,@\, d \Downarrow_r d_0 \quad \sigma \vdash \mathbf{map}\ p \,@\, s \Downarrow_r s_0}{\sigma \vdash \mathbf{map}\ p \,@\, \{d\} \cup s \Downarrow_r \{d_0\} \cup s_0}\text{(Map)}$$

$$\frac{\sigma \vdash p \,@\, d \Downarrow_r \mathsf{true}}{\sigma \vdash \mathbf{assert}\ p \,@\, d \Downarrow_r [\,]}\text{(Assert True)} \qquad \frac{\sigma \vdash p \,@\, d \Downarrow_r \mathsf{false}}{\sigma \vdash \mathbf{assert}\ p \,@\, d \Downarrow_r \mathbf{err}}\text{(Assert False)}$$

$$\frac{\sigma \vdash p_1 \,@\, d \Downarrow_r d_1}{\sigma \vdash p_1 \mid\mid p_2 \,@\, d \Downarrow_r d_1}\text{(OrElse 1)} \qquad \frac{\sigma \vdash p_1 \,@\, d \Downarrow_r \mathbf{err} \quad \sigma \vdash p_2 \,@\, d \Downarrow_r d_2?}{\sigma \vdash p_1 \mid\mid p_2 \,@\, d \Downarrow_r d_2?}\text{(OrElse 2)}$$

$$\frac{}{\sigma \vdash \mathbf{it} \,@\, d \Downarrow_r d}\text{(}\mathbf{it}\text{)} \qquad \frac{}{\sigma \vdash \mathbf{env} \,@\, d \Downarrow_r \sigma}\text{(}\mathbf{env}\text{)}$$

$$\frac{\sigma \vdash p_1 \,@\, d \Downarrow_r d_1 \quad \sigma \vdash p_2 \,@\, d_1 \Downarrow_r d_2?}{\sigma \vdash \mathbf{let\ it} = p_1\ \mathbf{in}\ p_2 \,@\, d \Downarrow_r d_2?}\text{(Let }\mathbf{it}\text{)}$$

$$\frac{\sigma \vdash p_1 \,@\, d \Downarrow_r \sigma_1 \quad \sigma + \sigma_1 \vdash p_2 \,@\, d \Downarrow_r d_2?}{\sigma \vdash \mathbf{let\ env} \mathrel{+}= p_1\ \mathbf{in}\ p_2 \,@\, d \Downarrow_r d_2?}\text{(Let }\mathbf{env}\text{)}$$

$$\frac{\sigma \vdash p_1 \,@\, d \Downarrow_r \sigma_1 \quad \neg\mathsf{compatible}(\sigma, \sigma_1)}{\sigma \vdash \mathbf{let\ env} \mathrel{+}= p_1\ \mathbf{in}\ p_2 \,@\, d \Downarrow_r \mathbf{err}}\text{(Let }\mathbf{env\ err}\text{)}$$

err propagation:

$$\frac{\sigma \vdash p \,@\, d \Downarrow_r \mathbf{err}}{\begin{array}{l} \sigma \vdash \oplus p \,@\, d \Downarrow_r \mathbf{err} \\ \sigma \vdash p \otimes p_2 \,@\, d \Downarrow_r \mathbf{err} \\ \sigma \vdash p_1 \otimes p \,@\, d \Downarrow_r \mathbf{err} \\ \sigma \vdash \mathbf{assert}\ p \,@\, d \Downarrow_r \mathbf{err} \\ \sigma \vdash \mathbf{let\ it} = p\ \mathbf{in}\ p_2 \,@\, d \Downarrow_r \mathbf{err} \\ \sigma \vdash \mathbf{let\ env} \mathrel{+}= p\ \mathbf{in}\ p_2 \,@\, d \Downarrow_r \mathbf{err} \end{array}}$$

Figure 3 CAMP semantics. $\boxed{\sigma \vdash p \,@\, d \Downarrow_r d?}$

the environment for evaluating p_2 (recall that "+" is defined only for compatible records). Incompatibility results in match failure.

Figure 3 also presents six rules enabling **err** propagation, written in compressed form. The antecedent of the given rule separately implies all six of the consequents.

To illustrate, consider the JRules pattern `M: Marketer(clients.contains(C.id));` from Line 4 in Figure 1. This corresponds to the CAMP pattern in Equation 1:

$$\mathbf{it}.\mathit{type} = \text{“}\mathsf{Marketer}\text{”} \wedge \mathbf{env}.C.\mathit{data}.\mathit{id} \in \mathbf{it}.\mathit{data}.\mathit{clients} \wedge \mathbf{let\ env} \mathrel{+}= [M : \mathbf{it}]\ \mathbf{in\ env} \qquad (1)$$

In words: if **it** is a record whose *type* attribute is "Marketer", and if the environment already has a binding for C with a *data.id* attribute that is an element of the bag **it**.*data.clients*, then bind **it** to M in the environment and return the enriched environment reified as a record.

3 Rules

This section describes CAMP Rules, a set of rules macros on top of the CAMP core calculus from Section 2, which we also implemented in the Coq proof assistant [16]. As shown in Figure 2, the rules macros bridge the gap between CAMP and real-world production rule languages such as JRules [24]. This section provides practical context for this work, describing the connection between CAMP rules and JRules.

3.1 JRules

This paper grew out of our work on IBM's "Operational Decision Manager: Decision Server Insights" product, or *ODM Insights* for short [29], a middleware for integrating event

processing with analytics. Figure 4 depicts the architecture of ODM Insights. Everything is driven by *events*, which are (possibly nested) objects in motion. When ODM Insights receives an incoming event, it routes it to an event processor. An *event processor* is a rule engine that can read individual entities from a shared store, and can also consult the result of analytics. An *entity* is a (possibly nested) object at rest, for instance, a JSON document [18]. The *store* holds all the entities. The event processor reacts to an event with two kinds of side effects: it can write to the store (to modify an existing entity or create a new entity), and it can send zero or more output events (also known as *actions*). On each firing, the event processor accesses at most a handful of entities. In contrast, the *analytics processor* periodically scans all the entities in the store to derive global insights useful to guide future actions. When the out-of-band analytics finish, the resulting insights become available so the reactive event processors can consult them.

Another important goal was to integrate the authoring experience: code for the event processor is written in JRules with extensions for spatiotemporal concepts. As state-of-the-art production rule technology did not offer a way to program the analytics, we needed to find a way to use production rules for queries over a large, and possibly distributed, data store. To fill this gap, this paper shows how to translate rules to NRA, for which there are known evaluation techniques over large stores.

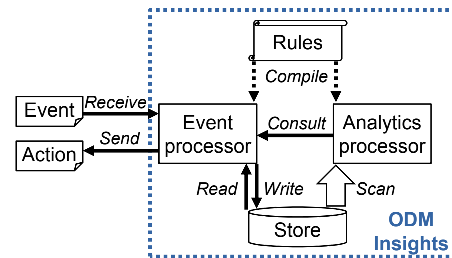


Figure 4 ODM Insights.

To demonstrate that CAMP accurately models a real-world production rule language, we implemented a compiler from JRules to CAMP rules. Our compiler reuses the front-end of the JRules compiler to get an abstract syntax tree, then traverses that tree to generate CAMP rules in Coq, along with a corresponding test harness. For a given rule, the test harness runs both the JRules engine and the translated Coq code against the same test input and compares the two results. With this technique, we tested several examples, to confirm that the JRules engine and the CAMP rules return the same result (modulo permutation over the collection of items being returned). While this testing does not have the rigor and completeness of mechanized proofs, it connects our mechanized theory to the real world.

3.2 Basic Production Rules in CAMP

Rules r are an intermediate language that bottoms out in CAMP patterns p from Definition 1. Rules r are defined via macros – meta-level functions that statically compute CAMP patterns p. For now, we will focus on basic rules drawn from the grammar $r ::= \boldsymbol{when}\ p; r \mid \boldsymbol{return}\ p$. Aggregation and negation are covered in Section 3.3.

The input to a JRules rule consists of a working memory. Our translation to CAMP encodes this as an environment mapping *WORLD* to a bag of working memory elements. We define the macro $\mathbf{WW}\,(p) := \mathbf{let\ it} = \mathbf{env}.\mathit{WORLD}\ \mathbf{in}\ p$ to apply a pattern over the working memory. Each working memory element is a typed object. We encode each such object as a record with two attributes, *type* (a string) and *data* (a record). This simple encoding of objects does not account for subtyping, but suffices for our purposes.

A production rule engine finds all ways in which a rule can match. We express this using a ***when*** macro, **map**ping each pattern over working memory as follows:

$$[\![\boldsymbol{when}\ p; r]\!] = \mathit{flatten}(\mathbf{WW}\,(\mathbf{map}\ (\mathbf{let\ env} \mathrel{+}= p\ \mathbf{in}\ [\![r]\!])))$$

In words: for each working memory element, attempt pattern p. If it succeeds, add the

```
1 when {
2    C: Client();
3    M: Marketer(
4         clients.contains(C.id));
5 } then {
6    insert new C2M(C, M);
7 }
```

Figure 5 JRules with no aggregation.

$$
\begin{array}{lll}
\textbf{\textit{when}} & & \mathbf{it}.type = \text{“Client”} \\
& \wedge & \mathbf{let\ env} \mathrel{+}= [C : \mathbf{it}]\ \mathbf{in\ env}; \\
\textbf{\textit{when}} & & \mathbf{it}.type = \text{“Marketer”} \\
& \wedge & \mathbf{env}.C.data.id \in \mathbf{it}.data.clients \\
& \wedge & \mathbf{let\ env} \mathrel{+}= [M : \mathbf{it}]\ \mathbf{in\ env}; \\
\textbf{\textit{return}} & & [type\ :\ \text{“C2M”}]* \\
& & [data\ :\ [client\ :\ \mathbf{env}.C]*\ [marketer\ :\ \mathbf{env}.M]]
\end{array}
$$

Figure 6 CAMP rule for Figure 5.

$$
\begin{aligned}
p_1 \wedge p_2 &:= \mathbf{let\ env} \mathrel{+}= \mathbf{assert}\ p_1\ \mathbf{in}\ p_2 \\
\mathbf{WW}\,(p) &:= \mathbf{let\ it} = \mathbf{env}.\mathit{WORLD}\ \mathbf{in}\ p \\
\mathbf{mapall}\ p &:= \mathbf{let\ env} \mathrel{+}= [x : \mathbf{map}\ p]\ \mathbf{in}\ \big((\#(\mathbf{env}.x) = \#\mathbf{it}) \wedge \mathbf{env}.x\big) \qquad x \text{ is } \mathbf{fresh} \\
\mathbf{mapsnone}\ p &:= \#(\mathbf{map}\ p) = 0
\end{aligned}
$$

Figure 7 Auxiliary definitions for rules.

resulting bindings to **env** and run the translated tail rule $[\![r]\!]$. This results in a bag of bags, where the inner bags are either empty for non-matches or singletons for matches. The final *flatten* returns the matches.

The output of a production rule consists of actions. We encode this in CAMP by returning a singleton bag containing the result of the action caused by a rule. This is done via a ***return*** macro: $[\![\textbf{\textit{return}}\ p]\!] = \{p\}$. The rule macros defined so far (**WW**, ***when***, ***return***, and $\wedge$ from Section 2.1) suffice to translate production rules without aggregation or negation. Figures 5 and 6 show a JRules program and the CAMP rule for it. Note that the `when` clause from JRules yields two ***when*** macros in the CAMP rule, each performing its own implicit loop over working memory. Since CAMP rules do not have side effects, we model the insertion by ***return***ing the computed value.

3.3 Rules with Aggregation and Negation

This section completes the language of rules macros from Section 3.2. Figure 7 presents definitions used in the semantics for rule macros. We already saw $\wedge$ and **WW**. The other two definitions, **mapall** and **mapsnone**, help encode aggregation and negation, respectively.

The **mapall** macro is similar to **map**, but ensures that the given pattern matches on all the data in the bag. Unlike **map**, which allows (and ignores) **err**s, **mapall** propagates any such **err**s. It does this by counting: if there are any **err**s, the result of **map** will be smaller than its input. To avoid recomputing the **map**, it stores its result in the environment, using a variable x that is **fresh**, meaning not used elsewhere in the program. The **mapsnone** macro also employs **map**. By counting to check that the resulting bag is empty it ensures that the given pattern does not match *any* data in the bag.

Definition 3 defines a rule as a sequence of patterns, each marked to indicate how the pattern should be interpreted. Patterns that are meant to (implicitly) be matched against each datum in working memory are signaled using ***when***. In contrast, ***global*** introduces patterns that should be run against the working memory itself (reified as a bag of data). Patterns marked with ***not*** are tested to ensure that they do not successfully match against *any* working memory data. A rule sequence is always terminated with ***return***.

$$\llbracket \textbf{\textit{when}}\ p; r \rrbracket = \mathit{flatten}\,(\mathbf{WW}\,(\mathbf{map}\ (\mathbf{let\ env} \mathrel{+}= p\ \mathbf{in}\ \llbracket r \rrbracket)))$$
$$\llbracket \textbf{\textit{global}}\ p; r \rrbracket = \mathbf{let\ env} \mathrel{+}= \mathbf{WW}\,(p)\ \mathbf{in}\ \llbracket r \rrbracket$$
$$\llbracket \textbf{\textit{not}}\ p; r \rrbracket = \mathbf{WW}\,(\mathbf{mapsnone}\ p) \wedge \llbracket r \rrbracket$$
$$\llbracket \textbf{\textit{return}}\ p \rrbracket = \{p\}$$

$$\frac{[\mathit{WORLD} : w] \vdash \llbracket r \rrbracket \mathbin{@} \mathit{nil} \Downarrow_r d?}{r \mathbin{@} w \Downarrow_r d?}\ \text{(Rule Evaluation)}$$

$$\mathbf{aggregate}(r, \oplus, p) := \mathbf{let\ it} = \llbracket r \rrbracket\ \mathbf{in}\ \oplus\mathbf{mapall}\ (\mathbf{let\ env} \mathrel{+}= \mathbf{it\ in}\ p)$$

Figure 8 Evaluating rules against working memory.

▶ **Definition 3** (Rules).

$$\text{(rules)} \quad r \quad ::= \quad \textbf{\textit{when}}\ p; r \mid \textbf{\textit{global}}\ p; r \mid \textbf{\textit{not}}\ p; r \mid \textbf{\textit{return}}\ p$$

Figure 8 presents a translation from rules to patterns. Rule evaluation proceeds by evaluating the translated pattern in an environment with *WORLD* bound to the desired working memory. Note that the translated pattern ignores the initial input, only accessing the working memory. Figure 8 also defines an **aggregate** macro with three parameters: a rule r, a reduction (unary) operator $\oplus$, and a transformer pattern p. First, the pattern for r is executed. Second, the result of r is transformed by **mapall**ing the transformer pattern p. Finally, the results of p are aggregated using $\oplus$. This is meant to be used with patterns marked ***global***, enabling nested aggregation over working memory. Consider the JRules code from Lines 4–5 of Figure 1: `Ms:aggregate {M:Marketer(clients.contains(C.id));} do collect{M};`. The CAMP rule for it is $\textbf{\textit{global}}[\mathit{Ms} : \mathbf{aggregate}(\textbf{\textit{when}}\ p_1; \textbf{\textit{return}}\ \mathbf{env}, \mathit{identity}, \mathbf{env}.M)];$, where p_1 is defined by Equation 1 at the end of Section 2.3. Note the use of ***when*** nested inside ***global*** to match all working memory elements again for aggregation.

4 CAMP to NRA

This section describes how to translate production rules to nested relational algebra (NRA). Since production rule languages support nesting both code and data, we use NRA which generalizes the well-known relational algebra (RA). In some cases, this generalization actually enables simplification. For instance, instead of RA's projection with attribute assignments, NRA offers a general map operator; and instead of RA's global table names, NRA can place tables with attributes of a top-level database record.

4.1 Nested Relational Algebra

We use the NRA from [14] with a bag semantics and extended with a conditional default operation. We consider a core set of queries, sufficient to express the full algebra. Richer queries (e.g., joins, unnest) can be built on top of this core.

▶ **Definition 4** (NRA syntax).

$$\begin{array}{lcl} \text{(queries)}\ q & ::= & d \mid \mathbf{In} \mid \oplus q \mid q_1 \otimes q_2 \mid \chi_{\langle q_2 \rangle}(q_1) \mid \sigma_{\langle q_2 \rangle}(q_1) \\ & \mid & q_1 \times q_2 \mid \bowtie^{d}{}_{\langle q_2 \rangle}(q_1) \mid q_1 \parallel q_2 \end{array}$$

In this grammar, d returns constant data, **In** returns the context value (usually a bag or a record), and $\oplus$ and $\otimes$ are unary and binary operators from Section 2.2. χ is the map operation on bags (in simple cases, map degenerates to conventional relational projection π), σ is selection, $\times$ is Cartesian product, and $\bowtie^d$ is the *dependent join* which evaluates q_2 with the context set one at a time to the results of q_1 and concatenates pairs of records resulting from q_1 and q_2. (Dependent join is written $q_1\langle q_2\rangle$ in [14] and MapConcat$\{q_2\}(q_1)$ in [31].) Sub-queries in subscripts with angle brackets $\langle ... \rangle$ are *dependent*, applied one at a time to the results of their sibling in the query plan.

The last operator, which we call *default*, is the only operator not originally proposed in [14], and is used to encode error propagation in CAMP. It evaluates its first operand and returns its value, unless that value is $\emptyset$, in which case it returns the value of its second operand (as default). For NRA to be equivalent to NNRC (a well-known correspondence [34]), some form of conditional must be included. Prior work included a similar default operation for null values, testing if the first data is null. Since we chose not to complicate our presentation with three-valued logic to implicitly propagate null values, we will instead use $\emptyset$ for errors.

4.2 Semantics

Figure 9 presents the semantics for NRA. The subscript $_a$ on the relation $\Downarrow_a$ stands for algebra. Previous work [14] used a denotational semantics; we chose a big-step operational semantics for consistency with CAMP. [1]

Evaluating a query q against input data d is written $q \,@\, d$. The **In** query returns that data, whereas the constant query returns the specified constant. Unary and binary operators evaluate the provided queries, and then evaluate the operator on the result(s).

A map query, $\chi_{\langle q_2\rangle}(q_1)$, is evaluated recursively using the two rules Map and Map $\emptyset$. The Map rule evaluates the query q_1, producing a bag s_1. One element is transformed using q_2 and the result is unioned with the result of mapping q_2 over the rest of s_1 (expressed as a constant query). The Map $\emptyset$ rule terminates this recursion with $\emptyset$.

Selection queries, $\sigma_{\langle q_2\rangle}(q_1)$, are evaluated recursively, similar to map queries. Each element in the bag produced by q_1 is kept only if applying the predicate q_2 returns true.

Product queries, $q_1 \times q_2$, require a double recursion. The Prod rule evaluates both q_1 and q_2 to produce bags s_1 and s_2. Elements d_1 and d_2 are chosen from them, and their product is unioned with the product of d_1 with the remainder of s_2 and the product of the remainder of s_1 with all of s_2.

Dependent join queries, $\bowtie^d{}_{\langle q_2\rangle}(q_1)$, first evaluate q_1 to produce a bag s_1. For each element d_1 of s_1, they evaluate q_2 with the context data set to d_1, yielding a bag s_2. The result is a bag consisting of all the record concatenations $d_1 \times d_2$ of records resulting from q_1 and q_2. Like product queries, dependent join queries use double recursion, with the main difference being that the inner recursion (over q_2) depends on the outer recursion (over q_1) by using its result as the context data.

Finally, the last query provided is the default query $q_1 \| q_2$. The rules for this query are straightforward, evaluating q_1 and looking at the result. If it is an empty bag, q_2 is evaluated and its result returned, otherwise the result of q_1 is returned.

[1] Our mechanization actually defines the semantics for all three languages via a computational denotational semantics. However, we chose to use a more conventional style for presentation.

$$\frac{}{d_0 \mathbin{@} d \Downarrow_a d_0}\ (\text{Constant}) \qquad \frac{}{\mathbf{In} \mathbin{@} d \Downarrow_a d}\ (\text{ID}) \qquad \frac{q \mathbin{@} d \Downarrow_a d_0 \quad \oplus d_0 = d_1}{\oplus q \mathbin{@} d \Downarrow_a d_1}\ (\text{Unary Operator})$$

$$\frac{q_1 \mathbin{@} d \Downarrow_a d_1 \quad q_2 \mathbin{@} d \Downarrow_a d_2 \quad d_1 \otimes d_2 = d_3}{q_1 \otimes q_2 \mathbin{@} d \Downarrow_a d_3}\ (\text{Binary Operator}) \qquad \frac{q_1 \mathbin{@} d \Downarrow_a \emptyset}{\chi_{\langle q_2 \rangle}(q_1) \mathbin{@} d \Downarrow_a \emptyset}\ (\text{Map } \emptyset)$$

$$\frac{q_1 \mathbin{@} d \Downarrow_a \{d_1\} \cup s_1 \quad q_2 \mathbin{@} d_1 \Downarrow_a d_2 \quad \chi_{\langle q_2 \rangle}(s_1) \mathbin{@} d \Downarrow_a s_2}{\chi_{\langle q_2 \rangle}(q_1) \mathbin{@} d \Downarrow_a \{d_2\} \cup s_2}\ (\text{Map})$$

$$\frac{q_1 \mathbin{@} d \Downarrow_a \{d_1\} \cup s_1 \quad q_2 \mathbin{@} d_1 \Downarrow_a \mathsf{true} \quad \sigma_{\langle q_2 \rangle}(s_1) \mathbin{@} d \Downarrow_a s_2}{\sigma_{\langle q_2 \rangle}(q_1) \mathbin{@} d \Downarrow_a \{d_1\} \cup s_2}\ (\text{Select True})$$

$$\frac{q_1 \mathbin{@} d \Downarrow_a \{d_1\} \cup s_1 \quad q_2 \mathbin{@} d_1 \Downarrow_a \mathsf{false} \quad \sigma_{\langle q_2 \rangle}(s_1) \mathbin{@} d \Downarrow_a s_2}{\sigma_{\langle q_2 \rangle}(q_1) \mathbin{@} d \Downarrow_a s_2}\ (\text{Select False})$$

$$\frac{q_1 \mathbin{@} d \Downarrow_a \emptyset}{\sigma_{\langle q_2 \rangle}(q_1) \mathbin{@} d \Downarrow_a \emptyset}\ (\text{Select } \emptyset) \qquad \frac{q_1 \mathbin{@} d \Downarrow_a \emptyset}{q_1 \times q_2 \mathbin{@} d \Downarrow_a \emptyset}\ (\text{Product } \emptyset_1) \qquad \frac{q_2 \mathbin{@} d \Downarrow_a \emptyset}{q_1 \times q_2 \mathbin{@} d \Downarrow_a \emptyset}\ (\text{Product } \emptyset_2)$$

$$\frac{q_1 \mathbin{@} d \Downarrow_a \{d_1\} \cup s_1 \quad q_2 \mathbin{@} d \Downarrow_a \{d_2\} \cup s_2 \quad \{d_1\} \times s_2 \mathbin{@} d \Downarrow_a s_3 \quad s_1 \times (\{d_2\} \cup s_2) \mathbin{@} d \Downarrow_a s_4}{q_1 \times q_2 \mathbin{@} d \Downarrow_a \{d_1 * d_2\} \cup s_3 \cup s_4}\ (\text{Prod})$$

$$\frac{q_1 \mathbin{@} d \Downarrow_a \{d_1\} \cup s_1 \quad q_2 \mathbin{@} d_1 \Downarrow_a \{d_2\} \cup s_2 \quad \bowtie^{d}_{\langle s_2 \rangle}(\{d_1\}) \mathbin{@} d \Downarrow_a s_3 \quad \bowtie^{d}_{\langle q_2 \rangle}(s_1) \mathbin{@} d \Downarrow_a s_4}{\bowtie^{d}_{\langle q_2 \rangle}(q_1) \mathbin{@} d \Downarrow_a \{d_1 * d_2\} \cup s_3 \cup s_4}\ (\text{DJ})$$

$$\frac{q_1 \mathbin{@} d \Downarrow_a \emptyset}{\bowtie^{d}_{\langle q_2 \rangle}(q_1) \mathbin{@} d \Downarrow_a \emptyset}\ (\text{DJ } \emptyset_1) \qquad \frac{q_1 \mathbin{@} d \Downarrow_a \{d_1\} \cup s_1 \quad q_2 \mathbin{@} d_1 \Downarrow_a \emptyset \quad \bowtie^{d}_{\langle q_2 \rangle}(s_1) \mathbin{@} d \Downarrow_a s_2}{\bowtie^{d}_{\langle q_2 \rangle}(q_1) \mathbin{@} d \Downarrow_a s_2}\ (\text{DJ } \emptyset_2)$$

$$\frac{q_1 \mathbin{@} d \Downarrow_a d_1 \quad d_1 \neq \emptyset}{q_1 \parallel q_2 \mathbin{@} d \Downarrow_a d_1}\ (\text{Default } \neg\text{Null}) \qquad \frac{q_1 \mathbin{@} d \Downarrow_a \emptyset \quad q_2 \mathbin{@} d \Downarrow_a d_2}{q_1 \parallel q_2 \mathbin{@} d \Downarrow_a d_2}\ (\text{Default Null})$$

Figure 9 NRA Semantics. $\boxed{q \mathbin{@} d \Downarrow_a d}$

4.3 Translating CAMP to NRA

There are two key mismatches between the evaluation of CAMP (introduced in Section 2) and the evaluation of NRA that must be addressed by any translation. CAMP is parameterized by both an environment and input data whereas NRA is parameterized only by input data. Additionally, the output of CAMP is part of the lifted domain $\mathcal{D}? = \mathcal{D} + \textbf{err}$, allowing any CAMP pattern to return a recoverable error. In contrast, NRA always returns data and has no concept of recoverable errors.

Figure 10 presents a compiler from CAMP to NRA that addresses both of these mismatches. The translation assumes (and preserves) a special encoding of both input and output to allow the semantics of a CAMP pattern to be expressed in NRA. The input of a compiled pattern is always a record with two components, E and D, storing the current environment and data. The output is always a bag. This bag is guaranteed to be either empty (representing a recoverable error) or a singleton of the data.

We will explain the translation starting with the simpler cases, not in the order presented. The rule for constants is trivial. The translations of **it** and **env** are a simple lookup, since they are both components of the input record.

Unary and binary operators ensure proper error propagation by taking advantage of the invariant that the returned data is a bag with zero or one elements. Mapping an operation over such a bag evaluates it on the data if present, and propagates the error otherwise

$$
\begin{aligned}
[\![d]\!] &= \{d\} \\
[\![\oplus p]\!] &= \chi_{\langle \oplus \mathbf{In} \rangle}([\![p]\!]) \\
[\![p_1 \otimes p_2]\!] &= \chi_{\langle \mathbf{In}.T_1 \otimes \mathbf{In}.T_2 \rangle}\big(\chi_{\langle [T_1:\mathbf{In}] \rangle}([\![p_1]\!]) \times \chi_{\langle [T_2:\mathbf{In}] \rangle}([\![p_2]\!])\big) \\
[\![\mathbf{map}\ p]\!] &= \big\{\mathit{flatten}\big(\chi_{\langle [\![p]\!] \rangle}\big(\rho_{D/\{T\}}\left(\{[E : \mathbf{In}.E] * [T : \mathbf{In}.D]\}\right)\big)\big)\big\} \\
[\![\mathbf{assert}\ p]\!] &= \chi_{\langle [\,] \rangle}\big(\sigma_{\langle \mathbf{In} \rangle}([\![p]\!])\big) \\
[\![p_1 \| p_2]\!] &= [\![p_1]\!] \| [\![p_2]\!] \\
[\![\mathbf{it}]\!] &= \{\mathbf{In}.D\} \\
[\![\mathbf{let\ it} = p_1\ \mathbf{in}\ p_2]\!] &= \mathit{flatten}\big(\chi_{\langle [\![p_2]\!] \rangle}\big(\rho_{D/\{T\}}\left(\{[E : \mathbf{In}.E] * [T : [\![p_1]\!]]\}\right)\big)\big) \\
[\![\mathbf{env}]\!] &= \{\mathbf{In}.E\} \\
[\![\mathbf{let\ env} \mathrel{+}= p_1\ \mathbf{in}\ p_2]\!] &= \mathit{flatten}\Big(\pi_{\langle [\![p_2]\!] \rangle}\big(\\
&\qquad \pi_{\langle [E:\mathbf{In}.E_2]*[D:\mathbf{In}.D] \rangle}\big(\\
&\qquad\quad \rho_{E_2/\{T_2\}}\big(\pi_{\langle \mathbf{In}*[T_2:\mathbf{In}.E+\mathbf{In}.E_1] \rangle}\big(\\
&\qquad\qquad \rho_{E_1/\{T_1\}}\left(\{\mathbf{In} * [T_1 : [\![p_1]\!]]\}\right)\big)\big)\big)\big)\Big) \\
\rho_{B/\{A\}}(q) &= \chi_{\langle \mathbf{In} - A \rangle}\Big(\bowtie^d_{\langle \chi_{\langle [B:\mathbf{In}] \rangle}(\mathbf{In}.A) \rangle}(q)\Big)
\end{aligned}
$$

Figure 10 Compiling CAMP to NRA.

(mapping $\emptyset$ to $\emptyset$). Binary operators store the two partial results in a record, then extract the components and apply the operator.

The translations of **assert** and orElse ($p_1 \| p_2$) take advantage of the translation mapping **err** to $\emptyset$. The selection operator is used for **assert**, along with a map that, in case of success, replaces `true` with the expected empty record.

To simplify the translation of the remaining patterns (**map**, **let it**, and **let env**), we introduce a derived operation, unnesting. $\rho_{B/\{A\}}(q)$ in Figure 10 unnests the nested bag stored in record attribute A, and renames its elements to attribute B. Given a bag $\big\{\big[A\colon \{a_1, ..., a_n\}, \overline{C_i\colon c_i}\big]\big\}$, unnest returns the bag $\big\{\big[B\colon a_1, \overline{C_i\colon c_i}\big], ..., \big[B\colon a_n, \overline{C_i\colon c_i}\big]\big\}$, using an intermediate bag $\big\{\big[B\colon a_1, A\colon \{a_1, ..., a_n\}, \overline{C_i\colon c_i}\big], ..., \big[B\colon a_n, A\colon \{a_1, ..., a_n\}, \overline{C_i\colon c_i}\big]\big\}$ created via a dependent join, then subtracting out the superfluous A attribute. This intermediate step is why we use a temporary name and have ρ rename as it unnests.

With unnesting in hand, let us look at the translation of **map** p from CAMP, which assumes that the current input data is a bag. In the NRA translation, that means that $\mathbf{In}.D$ is a bag. The compiler cannot, however, use the NRA map operation directly ($\chi_{\langle [\![p]\!] \rangle}(\mathbf{In}.D)$), as this would not preserve an important invariant of our translation: the input to a compiled pattern must be a record with the data and the environment. Instead, the map translation creates a singleton bag out of a record containing the environment (E) and the data bag. For the data it uses a temporary name, T. Unnesting allows us to obtain the required input data, a bag of records, each with the (appropriate part of the) data and the environment. This input can now be used with map (χ). Since the result is a bag of singleton bags, we finish the translation by flattening the result down to a bag of the data, and then lifting it back into a singleton bag (to preserve the output invariant).

The **let it** and **let env** translations use the unnesting operation to similar effect. Note that the translation of p_1 is a bag with at most one element, so the map accomplishes the

required sequencing and error propagation automatically. The **let env** rule needs to use unnesting twice: once to calculate and unnest the environment returned by p_1, and once to unnest the concatenation of that environment with the current environment. Note that this concatenation is done using the $+$ operation, which returns $\emptyset$ if the environments are not compatible. This is exactly the desired semantics, since $\emptyset$ in NRA represents **err** in CAMP.

4.4 Correctness

Theorem 5 asserts that the translation from CAMP to NRA in Figure 10 preserves semantics. A CAMP pattern that evaluates to d becomes an NRA query that evaluates to $\{d\}$ and a CAMP pattern that evaluates to **err** becomes an NRA query that evaluates to $\emptyset$.

▶ **Theorem 5** (Correctness of compiler from CAMP to NRA).

$$\sigma \vdash p \mathbin{@} d_1 \Downarrow_r d_2 \iff [\![p]\!] \mathbin{@} ([E:\sigma] * [D:d_1]) \Downarrow_a \{d_2\}$$
$$\sigma \vdash p \mathbin{@} d_1 \Downarrow_r \mathbf{err} \iff [\![p]\!] \mathbin{@} ([E:\sigma] * [D:d_1]) \Downarrow_a \emptyset$$

This theorem is verified by the accompanying mechanization. The proof is straightforward, relying on the invariants that the translation assumes and ensures. Due to the computational nature of our mechanized semantics, much of the apparent complexity of the proofs is reduced to simple computation. We also prove that given a CAMP pattern, the translated NRA query is at most a constant factor larger.

5 NRA to NNRC

Having introduced CAMP and a translation from CAMP to NRA, we now wish to go in the other direction, showing how to translate NRA back to CAMP. Rather than go directly from NRA to CAMP, we stage the translation through the named nested-relational calculus (NNRC), which provides a more declarative model and is well known to have the same expressivity as NRA [34]. This section introduces NNRC and a translation from NRA to NNRC. Section 6 completes the cycle, introducing a translation from NNRC back to CAMP.

Staging through NNRC allows us to split apart different aspects of the translation. As we will see in Section 5.3, all of the dependent queries in NRA get translated into a single NNRC construct, the comprehension. Section 6, which introduces a translation from NNRC back to CAMP, can then focus on the comprehension, without needing to separately handle all the dependent constructs provided by NRA. Taking this detour also allows us to mechanize the well-known equivalence between NRA and NNRC (staging the translation through CAMP).

5.1 Named Nested Relational Calculus

We assume a sufficiently large set of variables $\{x, y, ...\}$. The calculus is similar to the one used in [35], with a bag semantics.

▶ **Definition 6** (NNRC syntax).

$$(\text{exprs})\ e \quad ::= \quad x \mid d \mid \oplus e_1 \mid e_1 \otimes e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \ \{e_2 \mid x \in e_1\} \mid e_1\ ?\ e_2 : e_3$$

In this grammar, x is a variable, d is constant data and $\oplus$ and $\otimes$ are unary and binary operators, as defined in Section 2.2. The **let** expression allows for dependent sequencing: expression e_1 is evaluated and its result bound to x in the environment, which is then used to evaluate e_2. The bag comprehension $\{e_2 \mid x \in e_1\}$ first evaluates expression e_1, producing

$$\frac{\sigma(x) = d}{\sigma \vdash x \Downarrow_c d} \text{(Variable)} \qquad \frac{}{\sigma \vdash d \Downarrow_c d} \text{(Constant)} \qquad \frac{\sigma \vdash e \Downarrow_c d_0 \quad \oplus\, d_0 = d_1}{\sigma \vdash \oplus e \Downarrow_c d_1} \text{(Unary Operator)}$$

$$\frac{\sigma \vdash e_1 \Downarrow_c d_1 \quad \sigma \vdash e_2 \Downarrow_c d_2 \quad d_1 \otimes d_2 = d_3}{\sigma \vdash e_1 \otimes e_2 \Downarrow_c d_3} \text{(Binary Operator)}$$

$$\frac{\sigma \vdash e_1 \Downarrow_c d_1 \quad (x : d_1, \sigma) \vdash e_2 \Downarrow_c d_2}{\sigma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \Downarrow_c d_2} \text{(Let)} \qquad \frac{\sigma \vdash e_1 \Downarrow_c \emptyset}{\sigma \vdash \{e_2 \mid x \in e_1\} \Downarrow_c \emptyset} \text{(For } \emptyset\text{)}$$

$$\frac{\sigma \vdash e_1 \Downarrow_c \{d\} \cup s \quad (x : d, \sigma) \vdash e_2 \Downarrow_c d_0 \quad \sigma \vdash \{e_2 \mid x \in s\} \Downarrow_c s_0}{\sigma \vdash \{e_2 \mid x \in e_1\} \Downarrow_c \{d_0\} \cup s_0} \text{(For)}$$

$$\frac{\sigma \vdash e_1 \Downarrow_c \mathsf{true} \quad \sigma \vdash e_2 \Downarrow_c d_2}{\sigma \vdash e_1 \ ?\ e_2 : e_3 \Downarrow_c d_2} \text{(If True)} \qquad \frac{\sigma \vdash e_1 \Downarrow_c \mathsf{false} \quad \sigma \vdash e_3 \Downarrow_c d_3}{\sigma \vdash e_1 \ ?\ e_2 : e_3 \Downarrow_c d_3} \text{(If False)}$$

Figure 11 NNRC Semantics. $\boxed{\sigma \vdash e \Downarrow_c d}$

a bag, then expression e_2 is evaluated with x bound to the current element. The result of the comprehension is a bag of these results. The conditional $e_1\ ?\ e_2 : e_3$ first evaluates e_1; if the result is true, it evaluates e_2, otherwise it evaluates e_3.

5.2 Semantics

An environment σ is a mapping from a finite set of variables to values. We write $(x : d, \sigma)$ for the environment σ extended with variable x mapped to data d. Figure 11 describes the semantics of NNRC expressions using the judgment $\sigma \vdash e \Downarrow_c d$, meaning: under environment σ, expression e evaluates to d. The subscript $_c$ on the relation $\Downarrow_c$ stands for calculus. Unlike CAMP and NRA, expressions are not queries over input data, but are parameterized solely by their environment.

The rule for variables looks up the given variable in the environment and returns the associated data. Constant expressions return the given constant, irrespective of the environment. Unary and binary operator expressions evaluate the given expressions in the current environment, and then apply the given operator to the results.

Let expressions evaluate the first expression in the current environment and then evaluate the second expression in an environment enriched with a binding from the given variable to the result of evaluating the first expression.

Comprehensions, $\{e_2 \mid x \in e_1\}$, are similar to let expressions, except that e_1 returns a bag, and e_2 is evaluated with x bound to each element of that bag in turn. Rule For encodes this recursion, evaluating e_1 and then picking an element of the resulting bag and running e_2 on it. The result is unioned with the evaluation of a comprehension of e_2 over the remainder of the bag. Rule For $\emptyset$ enables this recursion to terminate.

The rules for the final type of expression, the conditional, are straightforward. The first expression is evaluated and its result used to determine which branch to evaluate.

5.3 Translation from NRA to NNRC

When compiling NRA queries to NNRC expressions, there are two main problems that need to be addressed: the different contexts and the need for variable names.

$$
\begin{aligned}
[\![d]\!]_x &= d \\
[\![\mathbf{In}]\!]_x &= x \\
[\![\oplus q]\!]_x &= \oplus [\![q]\!]_x \\
[\![q_1 \otimes q_2]\!]_x &= [\![q_1]\!]_x \otimes [\![q_2]\!]_x \\
[\![\chi_{\langle q_2 \rangle}(q_1)]\!]_x &= \{[\![q_2]\!]_t \mid t \in [\![q_1]\!]_x\} && t \text{ is } \textbf{fresh} \\
[\![\sigma_{\langle q_2 \rangle}(q_1)]\!]_x &= \mathit{flatten}(\{[\![q_2]\!]_t \ ?\ \{t\} : \emptyset \mid t \in [\![q_1]\!]_x\}) && t \text{ is } \textbf{fresh} \\
[\![q_1 \times q_2]\!]_x &= \mathit{flatten}(\{\{t_1 * t_2 \mid t_2 \in [\![q_2]\!]_x\} \mid t_1 \in [\![q_1]\!]_x\}) && t_1 \text{ is } \textbf{fresh} \wedge t_2 \text{ is } \textbf{fresh} \\
[\![\bowtie^d{}_{\langle q_2 \rangle}(q_1)]\!]_x &= \mathit{flatten}(\{\{t_1 * t_2 \mid t_2 \in [\![q_2]\!]_{t_1}\} \mid t_1 \in [\![q_1]\!]_x\}) && t_1 \text{ is } \textbf{fresh} \wedge t_2 \text{ is } \textbf{fresh} \\
[\![q_1 \parallel q_2]\!]_x &= \textbf{let } t = [\![q_1]\!]_x \textbf{ in } ((t = \emptyset)\ ?\ [\![q_2]\!]_x : t) && t \text{ is } \textbf{fresh}
\end{aligned}
$$

Figure 12 Compiling NRA to NNRC.

NRA queries and NNRC expressions run in different contexts. NRA queries do not have an environment, but are run against specified data. In contrast, NNRC expressions have only an environment and no other implicit data. The translation from NRA queries to NNRC expressions needs to store the input data for the query in the environment of the compiled expression. We parameterize the translation with the variable used for this mapping (subscript $_x$ in Figure 12).

The compiler also needs a way to manufacture variable names for use with NNRC let expressions and comprehensions. To simplify the presentation, we assume that we have a way to generate sufficiently fresh variables (variables that are distinct from any other variables used in the expression). We call such variables **fresh**. We revisit this in Section 5.4.

Figure 12 presents a compiler from NRA queries to NNRC expressions. Since NNRC provides declarative comprehensions, the compiler resembles a denotational semantics for NRA. The translation translates the identity query, **In**, as returning the value of x, the variable that holds the current data. The rest of the translation ensures that this association is preserved. Constants are translated to constants, and the sub-queries of unary and binary operator queries are translated and the specified operator reapplied.

The map query, $\chi_{\langle q_2 \rangle}(q_1)$, is expressed as a straightforward comprehension using a fresh variable t. The selection query $\sigma_{\langle q_2 \rangle}(q_1)$ is similarly translated to a comprehension. The body of the comprehension either returns a singleton bag containing the element of the bag returned by q_1 or $\emptyset$. The result is thus a bag of (empty or singleton) bags, which is flattened using the *flatten* operator.

Both the product $q_1 \times q_2$ and the dependent join $\bowtie^d{}_{\langle q_2 \rangle}(q_1)$ are expressed as two nested comprehensions. The resulting bag of bags is then flattened. The crucial difference between them is which data is used in the translation of q_2 in the inner comprehension. For the product, the same data, bound to x, is used. For the dependent join, the variable used in the outer comprehension is used, so the translation uses the current element of q_1 as the current data. This difference succinctly captures the dependency.

The final type of query, the default query, is translated into a conditional expression that inspects the result t of q_1, and evaluates q_2 if it is $\emptyset$ or returns t otherwise.

The usage of let expressions when translating default is the main reason we extend the traditional NNRC with let expressions. Existing work generally assumes that all the data is a bag. In that case, the let expression is not needed, since in that case we can express **let** $x = e_1$ **in** e_2 as $\mathit{flatten}(\{e_2 \mid x \in \{e_1\}\})$. However, that works only if we assume that e_2

returns a bag (otherwise the flatten may not even be well-typed). Since we did not want to limit the data our programs manipulate, we elected to add in let expressions to handle the case where the data may not be a bag.[2]

5.4 Correctness

The accompanying mechanization proves Theorem 7: our translation preserves semantics. NRA query q against data d evaluates to the same result (or lack thereof) as its translation into NNRC, $[\![q_1]\!]_x$, with an environment that associates x with the input d.

▶ **Theorem 7** (Correctness of compiler from NRA to NNRC)**.**

$$\textit{If } \sigma(x) = d_1 \textit{ then } q \mathbin{@} d_1 \Downarrow_a d_2 \iff \sigma \vdash [\![q]\!]_x \Downarrow_c d_2$$

The proof of Theorem 7 is straightforward, taking advantage of the previously mentioned similarity between the denotational semantics of NRA and the translation to NNRC. The proof essentially formalizes and verifies that correspondence.

The main additional challenge in the mechanization relates to the use of fresh variables. In Figure 12 we assumed the existence of **fresh**, which produces a "sufficiently unique" variable. In the mechanized semantics, this is formalized as a variety of freshness functions that produce variables fresh with respect to the relevant parts of sub-expressions and the input variable. The proof then verifies that each picked variable is indeed sufficiently fresh. Full details are included in the accompanying mechanization.

The mechanization also establishes that the compiler in Figure 12 produces a pattern at most a constant times larger than the input NRA. As a consequence, the compiler can clearly be observed to run in time polynomial in the input size.

6 NNRC to CAMP

This section completes the cycle from Figure 2, showing how to compile NNRC, as defined and presented in Section 5, back into the CAMP language from Section 2. This establishes the equivalence of all the languages, exhibiting compilers from CAMP to NRA to NNRC and back to CAMP. All these compilers cause at most a linear increase in code size, so the result of a full cycle of compilation yields a program semantically equivalent to the original and at most a constant times larger.

6.1 Translation from NNRC to CAMP

Figure 13 presents a compiler from NNRC to CAMP. Compilation is mostly straightforward, using CAMP's environment (**env**) as a target for NNRC's environment (σ). Since NNRC does not represent a query over data, the compilation does not need **it** except as part of the translation scheme. In particular, a compiled NNRC expression returns the same result regardless of the input data to the pattern (verified in the mechanization).

We need to be careful of shadowing (i.e., an inner variable definition hiding the previous definition for a variable with the same name), however, as the two languages' environments have different semantics when a variable appears twice. In CAMP, adding a variable to the environment ensures that its value is equal to the value of the previous binding for that

[2] Note that our compilation of CAMP to NRA (Section 4.3) actually enforces this restriction, since it lifts all data into a singleton bag. Nonetheless, we prefer the more general presentation.

$$
\begin{aligned}
[\![x]\!] &= \textbf{env}.x \\
[\![d]\!] &= d \\
[\![\oplus e]\!] &= \oplus[\![e]\!] \\
[\![e_1 \otimes e_2]\!] &= [\![e_1]\!] \otimes [\![e_2]\!] \\
[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!] &= \textbf{let it} = [\![e_1]\!] \textbf{ in let env} \mathrel{+}= [x : \textbf{it}] \textbf{ in } [\![e_2]\!] \\
[\![\{e_2 \mid x \in e_1\}]\!] &= \textbf{let it} = [\![e_1]\!] \textbf{ in mapall } (\textbf{let env} \mathrel{+}= [x : \textbf{it}] \textbf{ in } [\![e_2]\!]) \\
[\![e_1 \ ? \ e_2 : e_3]\!] &= \textbf{let env} \mathrel{+}= [x : [\![e_1]\!]] \textbf{ in} \qquad\qquad x \text{ is } \textbf{fresh} \\
&\quad ((\neg\neg\textbf{env}.x) \wedge [\![e_2]\!]) \parallel ((\neg\textbf{env}.x) \wedge [\![e_3]\!])
\end{aligned}
$$

Figure 13 Compiling NNRC to CAMP.

variable. To resolve this, we disallow shadowing, and assume that all comprehensions and let bindings in an NNRC expression use distinct variable names. It is always possible to rename the variables used in comprehensions and let expressions so that they are all distinct. This is verified by our mechanization.

Compiling variables, data, and operators is straightforward, using **env** to explicitly access the environment as needed. The **let** expression uses an outer **let it** to handle the required dependent sequencing, and then uses **let env** to add the calculated data to the environment, bound to the requested variable. As this variable is assumed unique (in the program), the compatibility clause of **let env** will always be satisfied.

Comprehensions ($\{e_2 \mid x \in e_1\}$) are compiled similarly to **let** sequencing expressions, except that **mapall**[3] is used to map $[\![e_2]\!]$ over each element in the collection produced by $[\![e_1]\!]$, with the variable x bound to the appropriate element.

Compiling the conditional expression $e_1 \ ? \ e_2 : e_3$ is slightly more complicated, as the pattern language only provides a (single-branched) orElse pattern ($p_1 \| p_2$), which handles errors. We use a **let env** pattern to evaluate and remember the compiled first expression. We assert that it evaluated to `true` and sequence the true branch e_2 with that assertion. If the assertion fails, that branch will not be executed. If this series of patterns fails, we use the orElse pattern to recover. In this case, either the initial assertion failed (e_1 either failed to match or did not return `true`), or e_2 failed. To distinguish among these possibilities, the recovery branch of the orElse first checks that the negation of (the remembered value of) e_1 is `true`. If it is `true`, then e_3 is evaluated. Since this branch ensures that e_1 evaluates to a Boolean (since the negation operation is defined only on Booleans), we use double negation to enforce that property for the first branch as well.

When using this let statement we need to be careful that we always pick a "fresh" (sufficiently unique) variable name. As we discussed earlier in the context of shadowing, the environment used by CAMP enforces that variables bound multiple times are all bound to the same data. This is not the semantics that we want. To avoid problems with duplicate variables in the source NNRC, we employed alpha-renaming. To avoid a similar type of problem in our compilation, we assume a way to generate a **fresh** variable name that does not conflict with any other variable name. This is used both in our translation of conditional expressions, and in our previous definition of **mapall**.

[3] Recall that **mapall** as well as the $\wedge$ operation were defined in Figure 7.

6.2 Correctness

The mechanization proves Theorem 8, verifying that the compiler is correct.

▶ **Theorem 8** (Correctness of compiler from NNRC to CAMP)**.** *Assuming that σ is well-formed (formalized in the accompanying mechanization):*

$$\sigma \vdash e \Downarrow_c d \iff \sigma \vdash [\![e]\!] \,@\, d_0 \Downarrow_r d$$

Note that d_0 is unconstrained, as a compiled NNRC expression does not access **it**.[4]

The proof is staged: we first prove correct a naive compiler that does not use **let env** or fresh variables, instead performing redundant computation. We then prove it produces a pattern semantically equivalent to the pattern produced by the compiler in Figure 13, where redundant computation is avoided to ensure the compiler preserves the original complexity. Semantics preservation of our compiler follows by composing these results.

Once more, formalizing these proofs requires a proper treatment of freshness. To do this, the mechanization formalizes the concept of free and bound variables, and provides a way to obtain a new variable that is not in a provided set. Ensuring that all required freshness conditions are met is a major challenge of the proofs and mechanization, requiring many well-formedness conditions to be defined on the environment and verified as preserved by the induction. Our mechanization also provides a method "unshadow" that renames variables in an NNRC expression to ensure that all bound variables are distinct (both within expression and from a provided environment). The mechanization proves that this transformation preserves semantics. Thus, our actual compiler runs "unshadow" and compiles the result, allowing all NNRC expressions to be compiled.

We have also mechanized a proof that the compiler in Figure 13 produces a pattern at most a constant times larger than the input NNRC. The compiler also clearly runs in time polynomial in the input size, although this property is not mechanized.

7 Type Checking

This paper has introduced a new language, CAMP, as well as formalized variants of existing languages, NRA and NNRC. All the semantics presented were partial; ill-typed programs may not evaluate. All of our compilers were careful to translate ill-typed programs into ill-typed programs. The correctness theorems all guarantee that a translation can evaluate successfully if and only if the original could evaluate successfully.

This section introduces types for data and operators (Section 7.1) and formalizes the type systems for CAMP (Section 7.2), NRA (Section 7.3), and NNRC (Section 7.4). Each type system is sound: well-typed programs always evaluate to a result. Note that in the case of CAMP, a recoverable error is a valid result of pattern matching. Furthermore, as formalized in Section 7.5, all of our compilers are type-preserving. Well-typed programs are compiled to well-typed programs (with an associated type) and vice versa.

Section 7.6 applies this result to type inference for the languages. In particular, this lets us build on related work [35] to equip CAMP with a polymorphic type inference algorithm, and prove that polymorphic type inference for CAMP is NP-complete.

[4] This is of course verified by the accompanying mechanization.

7.1 Types for Data Model and Operators

Our types for data include primitive types NIL, INT, BOOL, and STRING. The type of a (homogeneous) bag with elements of type τ is written $\{\tau\}$, punning the notation used for data bags. Similarly, $[\overline{A_i : \tau_i}]$ is the type of a record with attributes A_i, each with data of type τ_i. We use the notation $d :_d \tau$ to mean that data d has type τ.

As with data records, we define a notion of compatibility for record types. Two type records are considered compatible if any overlapping attributes have the same type. Note that two records can have compatible types but incompatible data, causing recoverable match failure. We define $*$ and $+$ as for data records: $[\overline{A_i : \tau_{A_i}}] * [\overline{B_j : \tau_{B_j}}]$ concatenates two record types, favoring the types of A's attributes in case of conflict. Compatible concatenation, $[\overline{A_i : \tau_{A_i}}] + [\overline{B_j : \tau_{B_j}}]$, also concatenates the record types, but is defined only if the two records are compatible.

The types of the unary operators, written $\oplus :_o \tau_1 \to \tau_2$, written here for a given value d of type τ, are as follows:

$$\begin{aligned} \mathit{identity}\, d &:_o \tau \to \tau & \mathit{flatten}\, d &:_o \{\{\tau\}\} \to \{\tau\} \\ \neg d &:_o \texttt{BOOL} \to \texttt{BOOL} & [A\!:\!d] &:_o \tau \to [A : \tau] \\ \{d\} &:_o \tau \to \{\tau\} & d.A &:_o [A : \tau, \overline{B_i : \tau_i}] \to \tau \\ \#d &:_o \{\tau\} \to \texttt{INT} & d{-}A &:_o [A : \tau, \overline{B_i : \tau_i}] \to [\overline{B_i : \tau_i}] \end{aligned}$$

The types of the binary operators, written $\otimes :_o \tau_1 \to \tau_2 \to \tau_3$, given here for two given values d_1 and d_2 of types τ_1 and τ_2 respectively, are as follows:

$$\begin{aligned} & d_1 = d_2 :_o \tau \to \tau \to \texttt{BOOL} \\ & d_1 \in d_2 :_o \tau \to \{\tau\} \to \texttt{BOOL} \\ & d_1 \cup d_2 :_o \{\tau\} \to \{\tau\} \to \{\tau\} \\ & d_1 * d_2 :_o [\overline{A_i : \tau_{A_i}}] \to [\overline{B_j : \tau_{B_j}}] \to [\overline{A_i : \tau_{A_i}}] * [\overline{B_j : \tau_{B_j}}] \\ & d_1 + d_2 :_o [\overline{A_i : \tau_{A_i}}] \to [\overline{B_j : \tau_{B_j}}] \to \{[\overline{A_i : \tau_{A_i}}] + [\overline{B_j : \tau_{B_j}}]\} \end{aligned}$$

7.2 CAMP Type System

Figure 14 presents a type system for CAMP. Where the CAMP semantics use a data environment σ, the CAMP type system uses a type context Γ. And where the CAMP semantics use an input datum **it**, the types of CAMP constructs are functions from an input type τ_0. We employ the same environment reification at the type level as we did for data, allowing the context Γ to be reified as a record type. The environment and type context can thus be related via a data typing relation, $\sigma :_d \Gamma$.

Most of the type rules are standard. We assume that **err** can be given any type, and so the rules do not need to account for recoverable errors, since **err** unifies with the type ascribed when a pattern does not have a recoverable error.

The Assert rule has an empty record type since **assert** returns an empty record if the assertion holds. The **env** rule reifies the context as a record type, just like the semantics reify the environment. The Let **env** rule checks the type of the second pattern with a context enriched by the bindings in the type record that type the first expression. Recall that the "+" operator ensures that these new bindings are compatible with the current context.

Theorem 9 asserts that the type system for CAMP given in Figure 14 is sound with respect to the semantics given in Figure 3. Given a well typed pattern, evaluating that pattern with an environment and input data that are appropriately typed will always yield a

$$\frac{d :_d \tau}{\Gamma \vdash d :_r \tau_0 \to \tau}\ \text{(Constant)} \qquad \frac{\Gamma \vdash p :_r \tau_0 \to \tau_1 \quad \oplus :_o \tau_1 \to \tau_2}{\Gamma \vdash \oplus p :_r \tau_0 \to \tau_2}\ \text{(Unary Operator)}$$

$$\frac{\Gamma \vdash p_1 :_r \tau_0 \to \tau_1 \quad \Gamma \vdash p_2 :_r \tau_0 \to \tau_2 \quad \otimes :_o \tau_1 \to \tau_2 \to \tau_3}{\Gamma \vdash p_1 \otimes p_2 :_r \tau_0 \to \tau_3}\ \text{(Binary Operator)}$$

$$\frac{\Gamma \vdash p :_r \tau_0 \to \tau_1}{\Gamma \vdash \mathbf{map}\ p :_r \{\tau_0\} \to \{\tau_1\}}\ \text{(Map)} \qquad \frac{\Gamma \vdash p :_r \tau_0 \to \mathtt{BOOL}}{\Gamma \vdash \mathbf{assert}\ p :_r \tau_0 \to [\,]}\ \text{(Assert)}$$

$$\frac{\Gamma \vdash p_1 :_r \tau_0 \to \tau_1 \quad \Gamma \vdash p_2 :_r \tau_0 \to \tau_1}{\Gamma \vdash p_1 \| p_2 :_r \tau_0 \to \tau_1}\ \text{(OrElse)} \qquad \frac{}{\Gamma \vdash \mathbf{it} :_r \tau_0 \to \tau_0}\ (\mathbf{it})$$

$$\frac{\Gamma \vdash p_1 :_r \tau_0 \to \tau_1 \quad \Gamma \vdash p_2 :_r \tau_1 \to \tau_2}{\Gamma \vdash \mathbf{let\ it} = p_1\ \mathbf{in}\ p_2 :_r \tau_0 \to \tau_2}\ \text{(Let } \mathbf{it}) \qquad \frac{}{\Gamma \vdash \mathbf{env} :_r \tau_0 \to \Gamma}\ (\mathbf{env})$$

$$\frac{\Gamma \vdash p_1 :_r \tau_0 \to [\overline{A_i : \tau_{A_i}}] \quad \left(\Gamma + [\overline{A_i : \tau_{A_i}}]\right) \vdash p_2 :_r \tau_0 \to \tau_2}{\Gamma \vdash \mathbf{let\ env} \mathrel{+}= p_1\ \mathbf{in}\ p_2 :_r \tau_0 \to \tau_2}\ \text{(Let } \mathbf{env})$$

Figure 14 Type rules for CAMP. $\boxed{\Gamma \vdash p :_r \tau_0 \to \tau_1}$

result – data or a recoverable error. If the result is data, then it will be appropriately typed. The proof proceeds by induction and is included in the mechanization.

▶ **Theorem 9** (Soundness of type system for CAMP).

$$\textit{if}\ \ (\Gamma \vdash p :_r \tau_0 \to \tau_1) \wedge (\sigma :_d \Gamma) \wedge (d_0 :_d \tau_0)\ \ \textit{then}\ \exists d_1?, (\sigma \vdash p \mathbin{@} d_0 \Downarrow_r d_1?) \wedge (d_1? :_d \tau_1)$$

7.3 NRA Type System

Figure 15 presents a type system for NRA (defined in Section 4.1). A well-typed query has an input type τ_0 and an output type τ_1, written $q :_a \tau_0 \to \tau_1$. Note how the difference between product and dependent join manifests as the difference in the input type of q_2. This is analogous to their different translations to NNRC in Figure 12.

The type system for NRA given in Figure 15 is sound with respect to the semantics given in Figure 9. Well typed queries, when applied to appropriately typed input data, will always evaluate successfully with appropriately typed output data. This is formalized by Theorem 10, which is verified in the accompanying mechanization.

▶ **Theorem 10** (Soundness of type system for NRA).

$$\textit{if}\ \ (q :_a \tau_0 \to \tau_1) \wedge (d_0 :_d \tau_0)\ \ \textit{then}\ \exists d_1, (q \mathbin{@} d_0 \Downarrow_a d_1) \wedge (d_1 :_d \tau_1)$$

7.4 NNRC Type System

Figure 16 presents a type system for the third language under discussion, NNRC (introduced in Section 5.1). This type system uses a type context to type an expression, similarly to CAMP. Unlike our system for CAMP, however, we do not need to reify contexts as records. Instead, we treat contexts as ordered lists. Adding x with type τ to the context Γ, written $(x : \tau, \Gamma)$, always succeeds, masking any previous binding for x. An NNRC environment σ has type Γ, written $\sigma : \Gamma$, if corresponding elements have the same variables, and the data in σ is typed by the corresponding type in Γ.

This type system for NNRC (Figure 16) is sound with respect to the semantics for NNRC given in Figure 11. Evaluating a well typed expression in an appropriately typed context

$$\frac{}{\mathbf{In} :_a \tau_0 \to \tau_0}\text{(ID)} \quad \frac{d :_d \tau}{d :_a \tau_0 \to \tau}\text{(Constant)} \quad \frac{q :_a \tau_0 \to \tau_1 \quad \oplus :_o \tau_1 \to \tau_2}{\oplus q :_a \tau_0 \to \tau_2}\text{(Unary Operator)}$$

$$\frac{q_1 :_a \tau_0 \to \tau_1 \quad q_2 :_a \tau_0 \to \tau_2 \quad \otimes :_o \tau_1 \to \tau_2 \to \tau_3}{q_1 \otimes q_2 :_a \tau_0 \to \tau_3}\text{(Binary Operator)}$$

$$\frac{q_1 :_a \tau_0 \to \{\tau_1\} \quad q_2 :_a \tau_1 \to \tau_2}{\chi_{\langle q_2 \rangle}(q_1) :_a \tau_0 \to \{\tau_2\}}\text{(Map)} \quad \frac{q_1 :_a \tau_0 \to \{\tau_1\} \quad q_2 :_a \tau_1 \to \texttt{BOOL}}{\sigma_{\langle q_2 \rangle}(q_1) :_a \tau_0 \to \{\tau_1\}}\text{(Select)}$$

$$\frac{q_1 :_a \tau_0 \to \{[\overline{A_i : \tau_{A_i}}]\} \quad q_2 :_a \tau_0 \to \{[\overline{B_j : \tau_{B_j}}]\}}{q_1 \times q_2 :_a \tau_0 \to \{[\overline{A_i : \tau_{A_i}}] * [\overline{B_j : \tau_{B_j}}]\}}\text{(Product)}$$

$$\frac{q_1 :_a \tau_0 \to \{[\overline{A_i : \tau_{A_i}}]\} \quad q_2 :_a [\overline{A_i : \tau_{A_i}}] \to \{[\overline{B_j : \tau_{B_j}}]\}}{\bowtie^d{}_{\langle q_2 \rangle}(q_1) :_a \tau_0 \to \{[\overline{A_i : \tau_{A_i}}] * [\overline{B_j : \tau_{B_j}}]\}}\text{(DJ)}$$

$$\frac{q_1 :_a \tau_0 \to \{\tau_1\} \quad q_2 :_a \tau_0 \to \{\tau_1\}}{q_1 \parallel q_2 :_a \tau_0 \to \{\tau_1\}}\text{(Default)}$$

Figure 15 Type rules for the Nested Relational Algebra. $\boxed{q :_a \tau_0 \to \tau_1}$

$$\frac{}{\Gamma \vdash x :_c \Gamma(x)}\text{(Variable)} \quad \frac{d :_d \tau}{\Gamma \vdash d :_c \tau}\text{(Constant)} \quad \frac{\Gamma \vdash e :_c \tau_1 \quad \oplus :_o \tau_1 \to \tau_2}{\Gamma \vdash \oplus e :_c \tau_2}\text{(Unary Operator)}$$

$$\frac{\Gamma \vdash e_1 :_c \tau_1 \quad \Gamma \vdash e_2 :_c \tau_2 \quad \otimes :_o \tau_1 \to \tau_2 \to \tau_3}{\Gamma \vdash e_1 \otimes e_2 :_c \tau_3}\text{(Binary Operator)}$$

$$\frac{\Gamma \vdash e_1 :_c \tau_1 \quad (x : \tau_1, \Gamma) \vdash e_2 :_c \tau_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 :_c \tau_2}\text{(Let)} \quad \frac{\Gamma \vdash e_1 :_c \{\tau_1\} \quad (x : \tau_1, \Gamma) \vdash e_2 :_c \tau_2}{\Gamma \vdash \{e_2 \mid x \in e_1\} :_c \{\tau_2\}}\text{(For)}$$

$$\frac{\Gamma \vdash e_1 :_c \texttt{BOOL} \quad \Gamma \vdash e_2 :_c \tau \quad \Gamma \vdash e_3 :_c \tau}{\Gamma \vdash e_1\ ?\ e_2 : e_3 :_c \tau}\text{(If)}$$

Figure 16 Type rules for the Named Nested Relational Calculus. $\boxed{\Gamma \vdash e :_c \tau}$

will always succeed with appropriately typed data. This is formalized by Theorem 11, which is verified in the accompanying mechanization.

▶ **Theorem 11** (Soundness of type system for NNRC).

$$\textit{if}\ \ (\Gamma \vdash e :_c \tau) \wedge (\sigma :_d \Gamma)\ \ \textit{then}\ \exists d, (\sigma \vdash e \Downarrow_c d) \wedge (d :_d \tau)$$

7.5 Type Preservation

As mentioned earlier, all our compilers are semantics preserving. In addition to preserving output data upon successful evaluation, they also preserve errors. A compiled program evaluates successfully if and only if the source program does.

All the compilers presented in this paper also preserve types. This is formalized in Theorem 12 and verified in the accompanying mechanization. This theorem asserts that the compilers preserve types in both directions: well-typed source programs guarantee well-typed compiled programs and well-typed compiled programs can only result from well-typed source programs. The forwards direction is the more traditional type preservation result, and ensures

that type-checking the source program suffices. We will discuss the backwards direction in Section 7.6.

▶ **Theorem 12** (Type Preservation). *Assuming that* Γ *is well-formed (formalized in the accompanying mechanization):*

$$\begin{array}{llll} \textit{CAMP}\leftrightarrow\textit{NRA}: & & \Gamma \vdash p :_r \tau_0 \to \tau_1 & \Leftrightarrow \quad [\![p]\!] :_a [E:\Gamma, D:\tau_0] \to \{\tau_1\} \\ \textit{NRA}\leftrightarrow\textit{NNRC}: & \textit{if } \Gamma(x)=\tau_0 \quad \textit{then} & q :_a \tau_0 \to \tau_1 & \Leftrightarrow \quad \Gamma \vdash [\![q]\!]_x :_c \tau_1 \\ \textit{NNRC}\leftrightarrow\textit{CAMP}: & & \Gamma \vdash e :_c \tau_1 & \Leftrightarrow \quad \Gamma \vdash [\![e]\!] :_r \tau_0 \to \tau_1 \end{array}$$

The accompanying mechanization formalizes the definition of well-formedness, which is needed to ensure that the context does not interfere with the variables introduced by the translations. We have chosen to omit these details from the paper, leaving them to the accompanying mechanization. The empty context is always well-formed.

For all of the compilations, when proving type preservation, it is helpful to prove that the typing rules are generally invertible. For the translation from CAMP to NRA, it is also helpful to derive an (invertible) type rule for the unnesting operator from Figure 10:

$$\frac{q : \tau_0 \to \{[A : \{\tau_A\}, \overline{C_i : \tau_{C_i}}]\}}{\rho_{B/\{A\}}\,(q) : \tau_0 \to \{[B : \tau_A, \overline{C_i : \tau_{C_i}}]\}}\ \text{(Unnest)}$$

In Section 6.1 we observed that the translation from NNRC to CAMP produces a pattern that ignores the input. This is apparent in the type preservation theorem, which allows the input type of an NNRC expression compiled to CAMP to be ascribed any type. This polymorphic type expresses that the CAMP pattern never looks at the input.

7.6 Type Inference

The results in Theorem 12 are bidirectional. The forwards direction is a typical statement of type preservation. As a statement about type checking, the backwards direction of these theorems are not very interesting. Why type check the results of a compilation when we could just type check the source? However, the bidirectionality of Theorem 12 allows us to connect not just type checking between the languages, but also type inference. Because our compilers all run in polynomial time and space, Theorem 12 ensures that type inference for one language can be used for another.

In particular, we can use this to build on related work and derive a polymorphic type inference algorithm for CAMP. Given a CAMP pattern p, we compile it to NRA using the compiler introduced in Section 4.3. We then compile the resulting NRA to NNRC using the compiler from Section 5.3. We then apply the polymorphic type inference algorithm for NNRC introduced by Van den Bussche and Vansummeren [35] and use Theorem 12 to recover the type of the original CAMP pattern. (Note that if a compilation from CAMP to NRA is well-typed, it must have a bag type, so the theorem indeed allows us to read any possible inferred type “backwards” to CAMP.)

Theorem 12 allows us to take advantage of Van den Bussche and Vansummeren’s NP-completeness result for polymorphic type inference [35], proving that polymorphic type inference for CAMP is NP-complete using a standard reduction argument. Starting with an NNRC expression, the compiler introduced in Section 6.1 can, in polynomial time, compile it to a CAMP pattern that is polynomial in the size of the original expression. Thus, any polynomial time polymorphic type inference algorithm for CAMP would yield a polynomial time polymorphic type inference algorithm for NNRC. Since polymorphic type inference for

NNRC is NP-complete, this means that polymorphic type inference for CAMP (and NRA) is also NP-complete.

8 Related Work

Production rule languages hark back to Forgy's seminal work on OPS5 [19]. OPS5 already has most of the features we formalize in CAMP, notably matching against a working memory using patterns in the context of a subject value and an environment of bound variables. Production rule systems are usually implemented with (variants of) the Rete algorithm [20], which relies on its own internal representation of rules and objects. Modern production rule languages and systems include Drools [5] and JRules [24], which extend the ideas from OPS5 with aggregation and support Java objects in working memory. In contrast to OPS5, Drools, and JRules, we show how to translate rules to NRA enabling the use of algorithms for scalable evaluation of aggregates [33], as well as execution over either a relational store or a map-reduce framework [8]. Recently proposed distributed extensions for rules languages [10, 30] do not consider support for aggregation and do not address scalability issues.

Datalog is without a doubt the most studied rules language in the database area and the relationship between Datalog and the relational algebra has been studied in depth [4]. As in production systems, Datalog relies on declarative logic-based rules, but with fixpoint semantics and restrictions (e.g., actions can only insert new facts) that guarantee termination. CAMP focuses on capturing production rules semantics, including pattern matching for complex objects, with negation and aggregation. Similarly to production rules, existing Datalog extensions included complex objects [2], negation [3], and aggregation [15, 28]. The .QL project at Semmle seems closest to our approach in that it includes both a type system [17] and investigates compilation into SQL for execution [32]. To the best of our knowledge, translations of (fragments of Datalog) to NRA have not been investigated and could represent an interesting direction for future work.

Our formalization uses the NRA originally developed in [1, 6, 14, 31] and NNRC [34, 35]. Cluet and Moerkotte show how to translate nested queries on object-oriented data into NRA [14] and extensions to relational optimization; Abiteboul and Beeri explore the expressive power of NRA [1], and Ré et al. translate XQuery to NRA [31]. While some of the details of NRA differ between these papers, they all extend the relational algebra with dependent operators and in particular a form of dependent join to handle nested data and queries. Tannen et al. introduce NNRC as a language for nested relations, and demonstrate its equivalence to NRA [34]. Van den Bussche and Vansummeren present a polymorphic type inference algorithm for NNRC [35], but not for NRA.

There have been few attempts at mechanized proofs for aspects of database languages and implementation [7, 12, 26]. Malecha et al. [26] and Benzaken et al. [7] mechanize the relational algebra in Coq. In contrast to our work, neither of these handles nesting nor formalizes a type system. Cheney and Urban formalize a subset of XQuery in Isabelle [12]. While they handle nesting and formalize a type system, they do not use relational algebra.

Many programming languages feature pattern matching. Patterns without nested objects are at the center of Prolog [13]. Pattern matching is also central to functional languages with algebraic data types. Haskell, for example, extends this with support for views extending matching to other types [37]. Matchete unifies matching on algebraic types with other pattern languages such as regular expressions [23]. Stratego matches an implicit datum for program transformation [36]. While algebraic types can nest and are thus closer to production rule languages, patterns in languages like Haskell scrutinize only one datum at a time. In contrast,

JMatch offers pattern matching for Java and combines it with iteration [25]. Thorn takes this further, allowing pattern matching in many control constructs [9]. However, those control constructs exist outside the pattern, less tightly integrated with matching as in production rule languages.

9 Conclusion

This paper introduces CAMP, a calculus that captures the essence of pattern matching and aggregation in production rule languages. It presents translations from CAMP to NRA to NNRC and back to CAMP, thus demonstrating that they are all equally expressive. This is important, because it shows a way to implement production rule languages over (nested) relational stores, while taking advantage of database techniques for efficient, distributed, and incremental execution. This paper includes theorems for correctness, type preservation, and size preservation of all translations, and the accompanying mechanization includes machine-checked proofs for all theorems. This not only validates our new results for CAMP, but also adds rigor to folklore results on NRA and NNRC that had not previously been mechanized.

References

1. Serge Abiteboul and Catriel Beeri. The power of languages for the manipulation of complex values. *Journal on Very Large Data Bases (VLDB J.)*, 4(4):727–794, 1995.
2. Serge Abiteboul and Stephane Grumbach. COL: A logic-based language for complex objects. In *Extending Database Technology (EDBT)*, pages 271–293, 1988.
3. Serge Abiteboul and Richard Hull. Data functions, Datalog and negation. In *International Conference on Management of Data (SIGMOD)*, pages 143–153, 1988.
4. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Addison-Wesley, 1995.
5. Michal Bali. *Drools JBoss Rules 5.0 Developer's Guide*. Packt Publishing, 2009.
6. Catriel Beeri and Yoram Kornatzky. Algebraic optimization of object-oriented query languages. *Theoretical Computer Science*, 116(1&2):59–94, 1993.
7. Véronique Benzaken, Evelyne Contejean, and Stefania Dumbrava. A Coq formalization of the relational data model. In *European Symposium on Programming (ESOP)*, pages 189–208, 2014.
8. K. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C-C. Kanne, F. Özcan, and E. Shekita. Jaql: A scripting language for large scale semistructured data analysis. In *Conference on Very Large Data Bases (VLDB)*, pages 1272–1283, 2011.
9. Bard Bloom and Martin Hirzel. Robust scripting via patterns. In *Dynamic Languages Symposium (DLS)*, pages 29–40, 2012.
10. Federico Cabitza, Marcello Sarini, and Bemardo Dal Seno. DJess – a context-sharing middleware to deploy distributed inference systems in pervasive computing domains. In *International Conference on Pervasive Services (ICPS)*, pages 229–238, 2005.
11. James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *International Conference on Functional Programming (ICFP)*, pages 403–416, 2013.
12. James Cheney and Christian Urban. Mechanizing the metatheory of mini-XQuery. In *Conference on Certified Programs and Proofs (CPP)*, pages 280–295, 2011.
13. William F. Clocksin and Christopher S. Mellish. *Programming in PROLOG*. Springer, 1987.

14 Sophie Cluet and Guido Moerkotte. Nested queries in object bases. In *Workshop on Database Programming Languages (DBPL)*, pages 226–242, 1993.

15 Mariano P. Consens and Alberto O. Mendelzon. Low complexity aggregation in GraphLog and Datalog. In *International Conference on Database Theory (ICDT)*, pages 379–394, 1990.

16 Coq reference manual, version 8.4pl41. `http://coq.inria.fr/`.

17 Oege de Moor, Damien Sereni, Pavel Avgustinov, and Mathieu Verbaere. Type inference for Datalog and its application to query optimisation. In *Principles of Database Systems (PODS)*, pages 291–300, 2008.

18 Miki Enoki, Jérôme Siméon, Hiroshi Horii, and Martin Hirzel. Event processing over a distributed JSON store: Design and performance. In *Web Information System Engineering (WISE)*, pages 395–404, 2014.

19 Charles L. Forgy. OPS5 user's manual. Technical Report 2397, CMU, 1981.

20 Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

21 Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *International Conference on Management of Data (SIGMOD)*, pages 328–339, 1995.

22 Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: Database-supported program execution. In *International Conference on Management of Data (SIGMOD)*, pages 1063–1066, 2009.

23 Martin Hirzel, Nathaniel Nystrom, Bard Bloom, and Jan Vitek. Matchete: Paths through the pattern matching jungle. In *Practical Aspects of Declarative Languages (PADL)*, pages 150–166, 2008.

24 IBM WebSphere ILOG JRules BRMS. `http://www.ibm.com/software/integration/business-rule-management/jrules-family/`.

25 Jed Liu and Andrew C. Myers. JMatch: Iterable abstract pattern matching for Java. In *Practical Aspects of Declarative Languages (PADL)*, pages 110–127, 2003.

26 J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *Principles of Programming Languages (POPL)*, 2010.

27 N. May, S. Helmer, and G. Moerkotte. Strategies for query unnesting in XML databases. *Transactions on Database Systems (TODS)*, 31(3):968–1013, 2006.

28 Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *Conference on Very Large Data Bases (VLDB)*, pages 264–277, 1990.

29 IBM Operational Decision Manager: Decision Server Insights. `http://www.ibm.com/support/knowledgecenter/SSQP76_8.7.0`.

30 Dana Petcu. Parallel Jess. In *International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 307–316, 2005.

31 Christopher Re, Jérôme Siméon, and Mary Fernandez. A complete and efficient algebraic compiler for XQuery. In *International Conference on Data Engineering (ICDE)*, 2006.

32 Damien Sereni, Pavel Avgustinov, and Oege de Moor. Adding magic to an optimising datalog compiler. In *International Conference on Management of Data (SIGMOD)*, pages 553–566, 2008.

33 Ambuj Shatdal and Jeffrey F. Naughton. Adaptive parallel aggregation algorithms. In *International Conference on Management of Data (SIGMOD)*, pages 104–114, New York, NY, USA, 1995.

34 Val Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In *International Conference on Database Theory (ICDT)*, pages 140–154, 1992.

35 Jan Van den Bussche and Stijn Vansummeren. Polymorphic type inference for the named nested relational calculus. *Transactions on Computational Logic (TOCL)*, 9(1), 2007.

36 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *Rewriting Techniques and Applications (RTA)*, pages 357–362, 2001.

37 Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages (POPL)*, pages 307–313, 1987.

Global Sequence Protocol: A Robust Abstraction for Replicated Shared State

Sebastian Burckhardt[1], Daan Leijen[1], Jonathan Protzenko[1], and Manuel Fähndrich[2]

1 **Microsoft Research, USA**
2 **Google, USA**

Abstract

In the age of cloud-connected mobile devices, users want responsive apps that read and write shared data everywhere, at all times, even if network connections are slow or unavailable. The solution is to replicate data and propagate updates asynchronously. Unfortunately, such mechanisms are notoriously difficult to understand, explain, and implement.

To address these challenges, we present GSP (global sequence protocol), an operational model for replicated shared data. GSP is simple and abstract enough to serve as a mental reference model, and offers fine control over the asynchronous update propagation (update transactions, strong synchronization). It abstracts the data model and thus applies both to simple key-value stores, and complex structured data. We then show how to implement GSP robustly on a client-server architecture (masking silent client crashes, server crash-recovery failures, and arbitrary network failures) and efficiently (transmitting and storing minimal information by reducing update sequences).

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases distributed computing, eventual consistency, GSP protocol

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.568

1 Introduction

Many applications can benefit from replicating shared data across devices, because it is often desirable to keep applications responsive even if network connections are slow or unavailable. Unfortunately, the CAP theorem [4, 17, 19] shows that strong consistency (such as linearizability or sequential consistency) requires communication with a reliable server or with a majority partition on each update, which becomes slow or impossible if network connections are slow or unavailable. Since responsiveness is often more important than strong consistency, researchers and practitioners have proposed the use of various forms of eventual consistency [8, 9, 15, 21, 29]. In such systems, update propagation and conflict resolution is lazy, proceeding only when network conditions permit, and replicas may temporarily differ, while converging to the same state eventually.

Although asynchronous update propagation and eventual consistency offer clear benefits, they are also more difficult to understand, both for system implementors and client programmers, motivating the need for simple programming models.

Previous work on replicated data types [7, 23, 24] and cloud types [5, 14] suggests that higher-level data abstractions can mitigate the mental overhead of working with weakly consistent replicas, as they can resolve conflicts automatically and prevent us from accidentally breaking representation invariants due to unexpected races.

© Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich;
licensed under Creative Commons License CC-BY

To evaluate these ideas in practice, we have implemented the cloud types model [5, 9, 14] in a scripting language for mobile devices. Our experiences suggest that it can indeed provide significant benefits. In particular, the automation of communication, error handling, and replication substantially simplifies the app development. However, data abstraction is not enough, and there remains room for improvement in several aspects:

- *Reasoning.* Client programmers often misunderstand where exactly they risk consistency errors, erring both on the safe and the unsafe side. Moreover, they are generally wary of "magical solutions" that do not convey an intelligible mechanism. Above all, what they need is a simple mental reference model to understand how to use the mechanism appropriately to write correct programs.

  The existing consistency models for cloud types are either too abstract for non-experts in memory consistency (e.g. the axiomatic model in citeect), or too complicated and overly general for the situation at hand (e.g. the revision diagram model in [5, 9]).
- *Judicious Synchronization.* While asynchronous update propagation is sufficient in most situations, for many apps we encountered a few situations where strong synchronization is needed (e.g. finalizing a reservation, ending an auction, or joining a game with an upper limit on the number of players). Thus, it is important that programmers can easily choose between synchronous and asynchronous reads and updates (and pay the cost of synchronicity only when they ask for it).
- *Robust Implementations.* Implementations must be carefully engineered to hide failures of clients, the network, and the server, and to minimize the amount of data stored and transmitted.

  For example, The cloud types implementations in [5, 14] do not discuss failures of any kind, and transmit the entire state in each message, which is impractical unless the amount of data shared is small. Moreover, the pushing and pulling of updates between client and server cannot proceed concurrently but is forced to alternate, which introduces significant delays.

In this paper, we describe several improvements in these areas. Specifically, we make the following contributions:

- We introduce the global sequence protocol (GSP), an operational model describing the system behavior precisely, yet abstractly enough to be suitable as a simple mental model of the replicated store. It is based on an abstract *data model* that can be instantiated to any data type, be it a simple key-value store, or the rich cloud types model. We compare GSP to the TSO (total store order) memory model and discuss its consistency properties.
- We show how GSP supports judicious use of synchronization. *Push* and *pull* operations give programmers precise control over the update propagation, and *flush* allows them to perform reads and updates synchronously whenever desired, thus recovering strong consistency.
- We present a detailed system implementation model of GSP that provides significant advantages:

  Robust Streaming. Updates are streamed continuously in both directions between server and client. We show precisely how clients may crash silently, how the server may fail and recover, how connections are established, how they can fail, and how they can be reconnected and resume transmission correctly, without disrupting the execution of the client program at any point.

  Reduction. Update sequences often exhibit redundancy (for example, if a variable is assigned several times, only the last update matters). We show how to eagerly reduce

update sequences, storing them in reduced form in state or delta objects. This means that our implementation stores and propagates a minimal amount of information only.
- We have implemented the ideas presented in this paper as an extension of TouchDevelop, a freely available programming language and development environment. Thus, we have made the cloud types programming model publicly available online for inspection and experimentation, and we provide links to a dozen example applications.

Overall, our work marks a big step forward towards a credible programming model for automatically replicated, weakly consistent shared data on devices, by providing both an understandable high-level system and data model, and a robust implementation containing powerful and interesting optimizations.

2 Overview

To write correct programs, we need a simple yet precise mental model of the store. In a conventional setting, we assume a single-copy semantics where client programs can read and write the shared data atomically. But to tolerate slow and unreliable connections, we must find an alternative model that accounts for the existence of multiple copies, i.e. multiple versions of the shared data.

To this end, we introduce in this paper an operational model of a replicated store, called *Global Sequence Protocol* (GSP). It is based on the simple idea that clients eventually agree on a global sequence of updates, while seeing a subsequence of this final sequence at any given point of time.

We introduce GSP in four stages. First, we clarify how to abstract the data operations (Section 3). Then, we introduce and explain Core GSP, a basic version of GSP that does not include transactions or synchronization (Section 4), and discuss various aspects of its consistency model. In Section 5 we present transactional GSP, and explain the benefits of its transactions and synchronization support. In Section 6, we show in detail how the GSP protocol can be realistically implemented on a client-server topology in a way that transparently hides channel failures, silent crashes of clients, and crash-recovery failures of the server. A cornerstone of the implementation is the use of *reduction*, which eliminates redundancy from update sequences.

We then conclude the paper by reporting on our practical experiences with implementing and operating GSP as an extension of TouchDevelop (Section 7), and comparing with related work (Section 8).

3 Data Models

In our experience, the key mental shift required to understand replicated data is to understand program behavior as a sequence of updates, rather than states. To this end, we characterize the shared data by its set of updates and queries, and represent a state by the sequence of updates that have led to it.

Sequence notations. We write T^* for the type of sequences of type T. Furthermore, [] is the empty sequence, $s_1 \cdot s_2$ is the concatenation of two sequences, $s[i]$ is the element at position i (starting with 0), and for a nonempty sequence s ($s.length > 0$), the expression $s[1..]$ denotes the subsequence satisfying $s = s[0] \cdot s[1..]$.

Rather than fixing the set of update and read operations upfront, we represent them using abstract types for updates, reads, and values.

```
abstract type Update, Read, Value;
```

Likewise, we abstractly represent the semantics of operations by a function *rvalue* that takes a read operation and a sequence of updates, and returns the value that results from applying all the updates in the sequence to the initial state of the data:

function $rvalue : Read \times Update^* \rightarrow Value$

We call a particular binding for *Update*, *Read*, *Value* and *rvalue* a *data model*.

3.1 Examples

Register. We can define a data model for a *register* using the following update and read operations

$$Update = \{ wr(v) \mid v \text{ in } Value \}$$
$$Read = \{ rd \}$$

and define the value returned by a read operation to be the last value written:

$$rvalue(rd, s) = \text{match } s \text{ with}$$
$$[] \rightarrow \text{undefined}$$
$$s_0 \cdot wr(v) \rightarrow v$$

Counter. We can define a data model for a *counter* as follows:

$$Update = \{ inc \}$$
$$Read = \{ rd \}$$
$$rvalue(rd, s) = s.length$$

where a read simply counts the number of updates.

Key-value Store. Perhaps the most widely used data type in cloud storage is the *key-value store*, which will serve as our main running example. We can define its data model as follows:

$$Update = \{ wr(k,v) \mid k,v \text{ in } Value \}$$
$$Read = \{ rd(k) \mid k \text{ in } Value \}$$

$$rvalue(rd(k), s) = \text{match } s \text{ with}$$
$$[] \rightarrow \text{undefined}$$
$$s_0 \cdot wr(k_0,v) \rightarrow \text{if } (k = k_0) \text{ then } v \text{ else } rvalue(rd(k), s_0)$$

Reduction of Update Sequences. For readers who may be alarmed by the prospect of having to store and transmit long update sequences: note that we will introduce state and delta objects in Section 6.1, which store update sequences in reduced form (for example, a key-value store needs to store only the last update for a given key).

4 Core GSP

We show a basic version of the global sequence protocol in Fig. 1, which includes the data operations (reads and updates), but omits synchronization and transactions for now.

The protocol specifies the behavior of a finite, but unbounded number of clients, by defining the state of each client, and transitions that fire in reaction to external stimuli. The transitions fall into two categories: the interface to the client program (from where update and read operations arrive), and the interface to the network (from where messages arrive).

The clients communicate using *reliable total order broadcast* (RTOB), a group communication primitive that guarantees that all messages are reliably delivered to all clients, and in the same total order. RTOB has been well studied in the literature on distributed systems [12, 16],

```
role Core_GSP_Client {

  known : Round* := []; // known prefix of global sequence
  pending : Round* := []; // sent, but unconfirmed rounds
  round : ℕ := 0; // counts submitted rounds

  // client program interface
  update(u : Update) {
    pending := pending · u;
    RTOB_submit( new Round { origin = this, number = round++ , update = u } );
  }
  read(r : Read) : Value {
    var compositelog := known · pending;
    return rvalue(r, updates(compositelog));
  }

  // network interface
  onReceive(r : Round) {
    known := known · r;
    if (r.origin = this) {
      assert(r = pending[0]); // due to RTOB total order
      pending := pending[1..];
    }
  }

  // rounds data structure
  class Round { origin : Client, number : ℕ, update : Update }
  function updates(s : Round*) : Update* { return s[0].update · · · s[s.length - 1].update; }
}
```

Figure 1 Core Global Sequence Protocol (GSP).

and is often used to build replicated state machines. It can be implemented on various topologies (such as client-server or peer-to-peer) and for various degrees of fault-tolerance. We describe one particular such implementation and important optimizations in Section 6. Core GSP stores and propagates updates as follows.

- Each client stores a currently known prefix of the global update sequence in *known*, and a sequence of pending updates in *pending*.
- When the client program issues an update, we (1) append this update to the sequence of pending updates, and (2) wrap the update into a *Round* object, which includes the origin and a sequence number, and broadcast the round.
- When receiving a round, we append the contained update to the known prefix of updates. Moreover, if this round is an echo (it originated on the same client), we remove it from the pending queue.

Since RTOB delivers messages in the same order to all recipients, the *known* prefixes in the clients (while not necessarily the same length at any given time) always match. Also, an echo of a round always matches the first (oldest) element of the *pending* queue.

When a client issues a read, we combine the update sequences in the *known* prefix and in the *pending* operations to determine the value returned by the read. Thus, it appears to the client program that its own updates have taken effect, before they are confirmed (i.e. before they are processed and echoed by the RTOB). This consistency property is sometimes called Read-my-Writes [28]).

An important point is that we cannot rely on RTOB being fast: at best, it requires a server roundtrip, and at worst, it can be stalled for prolonged periods by a failure or by a network partition, for example if the client is offline. Thus, making the updates in the pending queue visible to reads is essential for applications to appear responsive.

▶ **Example 1.** We can implement a causally consistent key-value store by using the read $rd(k)$ and write $wr(k,v)$ operations defined earlier. Then clients can always read and write any key without waiting for communication. In particular, the store remains operational even on clients that are temporarily offline. If two clients write a different value for the same key, they may temporarily see a different value, but once both updates have gone through the RTOB, their relative order in the global sequence determines the final value: the last writer wins.

4.1 Beware Consistency

By design, Core GSP is not strongly consistent: *updates are asynchronous and take effect with a delay.* Programmers who are not aware of this can easily run into trouble. For example, consider a program that tries to increment a value for a given key by reading it, adding one, and then writing it back:

$\mathsf{var}\ x = rd(\text{“counter”})$
$wr(\text{“counter”}, x + 1)$

This counter implementation does not count correctly if called concurrently. For example, two readers may both read the current value 0, and then both issue an update wr(“counter”, 1). Thus, the final value (once both updates have gone through RTOB) is 1, not 2 as we would like.

We show in Section 5 how to extend Core GSP with synchronization operations that can be used to enforce strong consistency where needed (at the expense of requiring communication, and losing the benefit of offline availability).

However, in many cases, there is a more elegant solution that avoids expensive synchronization. The trick is to use a richer data model that lets us express the update directly, at a higher level. For example, if the data model supports updates of the form $add(k,v)$, we can increment a counter by calling

$add(\text{“counter”}, 1)$

which always counts correctly: all add operations appear in the global sequence, and the read operation can correctly accumulate them. In general, the idea of including application-specific update operations in the data model is a powerful trick that can help to avoid synchronization in many situations.

4.2 Cloud Types

Although key-value stores are a powerful primitive, they are cumbersome and error-prone to work with directly. Productivity is greatly aided by a capability to declare structured data with richer update and query semantics.

Luckily, it turns out we can quite easily define higher level data types on top of the data model abstraction (section 3). In particular, we can implement full Cloud Types as proposed by previous work [5, 14]. Cloud types allow users to define and compose all of the data type examples given earlier (registers, counters, key-value stores), plus tables, which support dynamic creation and deletion of storage.

Cloud types also help to mitigate consistency issues, since concurrent updates are handled in a way that is consistent with the semantics of the type. For example, all integer-typed fields support an $add(n)$ operation.

In the extended technical report [11] we show how to define cloud types as a data model, and how to implement state and delta objects that optimally reduce the update sequences.

4.3 Eventual Consistency

Although GSP does not provide strong consistency, its consistency guarantees are still as strong as possible for a protocol that remains available under network partitions: it is quiescently consistent, eventually consistent, and causally consistent (as defined in [8], for example).

It is *quiescently consistent*, because when updates stop, clients converge to the same *known* prefix with empty *pending* queue (this is the original definition of eventual consistency as introduced in [29]). It is *eventually consistent* (as defined in [6, 9]) because all updates become eventually visible to all clients, and are ordered in the same arbitration order. It is *causally consistent* because an update U by some client C cannot become visible to other clients before all of the updates (let's call them V) that are visible to client C at the time it performs U. The reason is that the updates V consist of (1) the common server prefix, or (2) pending updates, which are all guaranteed to become visible to other clients no later than U.

Comparison to TSO. Core GSP appears conceptually (and even in name) quite similar to TSO [31] (total store order), a widely used relaxed memory model that queues stores performed by a processor in a local store buffer, from where they drain to memory asynchronously. This naturally leads us to ask the question: is Core GSP observationally equivalent to TSO? Interestingly, the answer depends on the notion of observational equivalence. If we assume that the relative order of operations on different clients is not directly observable (which is a common assumption for memory models, where clients are processors that do not communicate directly), the two are indeed equivalent. However, if the relative order of operations on different clients is observable (which is a reasonable assumption for distributed interactive applications with external means of communication), then they are not equivalent, as the following scenario illustrates.

Consider that the key-value store data model represents shared memory in a multiprocessor, which initially stores 0 for each address, and consider two clients performing the following interleaving of operations (where each column shows the operation of one client, and vertical placement defines the observed interleaving of the operations):

$wr(A,2)$	.
.	$wr(B,1)$
.	$wr(A,1)$
.	$rd(A) \rightarrow 2$
$rd(B) \rightarrow 0$	.

This interleaving is not observable on TSO: since the client on the right sees $rd(A)$ return 2, it must be the case that $wr(A,2)$ has drained to memory after $wr(A,1)$ drained. Since writes by the same client drain to memory in order, this implies that $wr(B,1)$ must have drained

to memory sometime before $rd(A)$ returns, and thus before $rd(B)$ is called, thus the $rd(B)$ cannot return 0. However, under GSP, this interleaving is possible, because $rd(B)$ may be called before the RTOB delivers the update for $wr(B,1)$ to the client on the left.

5 Transactional GSP

The Core GSP protocol we introduced in the previous section is already quite useful. However, it can be further improved by adding support for transactions and synchronization.

In this section, we introduce transactional GSP, which adds the synchronization operations *push* and *pull*, and the synchronization query *confirmed*. These additions give the programmer more control. The transactional GSP protocol is shown in Fig. 2. It is derived from Core GSP (Fig. 1), but improves the design in the following three aspects.

Update Transactions. Often, a client program updates several data items at a time, and those updates are meant to be atomic. For example,

wr("items", "[key1,key2]")
wr("key1", "something")
wr("key2", "something else")

In the global sequence model, the updates may arrive at another client at different times, thus that client may see an intermediate state that it was not supposed to observe.

To solve this problem, GSP uses a *transactionbuffer*. Updates performed by the client program go into this buffer. All updates in the buffer are broadcast in a single round when the client program calls *push*, and only then. They effectively form an 'update transaction' that is persisted and transmitted atomically. Updates in the transactionbuffer are included in the composite log, thus they are immediately visible to subsequent reads.

Read Stability. In the global sequence protocol, an update arriving from the network can interleave in unpredictable ways with the locally executing client program. In particular, if a client program performs two reads in a sequence, the second read may return a different value. In our experience, it is very difficult to write correct programs under such conditions (cf. data races in multiprocessor programs).

To solve this problem, GSP uses a *receivebuffer*. Received rounds are stored in this buffer. All rounds in the *receivebuffer* are processed when the client program calls *pull*, and only then. Thus, the client program can rely on read stability – the visible state can change only when issuing *pull*, or when performing local updates.

Confirmation Status. It is often desirable to find out if an update has committed (i.e. is now part of the global sequence constructed by the RTOB). Since this is impossible for client programs to detect in the global sequence protocol, we add a new function *confirmed* to the interface which returns true iff there are no local updates awaiting confirmation.

5.1 Discussion

We now discuss several interesting aspects of the transactional GSP model related to consistency and synchronization.

```
role GSP_Client {
  known : Round * := []; // known prefix of global sequence
  pending : Round * := []; // sent, but unconfirmed rounds
  round : ℕ := 0; // counts submitted rounds
  transactionbuffer : Update * := [];
  receivebuffer : Round * := [];

  // client program interface
  update(u : Update) { transactionbuffer := transactionbuffer · u; }
  read(r : Read) : Value {
    var compositelog := updates(known) · updates(pending) · transactionbuffer;
    return rvalue(r, compositelog);
  }
  confirmed() : boolean { return pending = [] && transactionbuffer = [] }
  push() {
    var r := new Round { origin = this, number = round ++ , updates = transactionbuffer};
    tob_submit( r );
    pending := pending · r;
    transactionbuffer := [];
  }
  pull() {
    foreach(var r in receivebuffer) {
      known := known · r;
      if (r.origin = this) { pending := pending[1..]; }
    }
    receivebuffer := [];
  }

  // network interface
  onReceive(r : Round) { receivebuffer := receivebuffer · r; }

  // rounds data structure
  class Round { origin : Client, number : ℕ, updates : Update }
  function updates(s : Round *) : Update * { return s[0].updates · · · s[s.length - 1].updates;}
}
```

Figure 2 Transactional GSP (Global Sequence Protocol).

On-Demand Strong Consistency. GSP is sufficiently expressive to allow client programs to recover strong consistency when desired. To this end, we can write a flush operation that waits for all pending updates to commit (and receives any other updates in the meantime):

```
flush() {
  push();
  while (! confirmed()) { pull(); }
}
```

Using *flush*, we can implement linearizable (strongly consistent) versions of any read operation r or update operation u as follows:

$synchronous_update(u)$ { $update(u)$; $flush()$; }
$synchronous_read(r)$ { $flush()$; $read(r)$; }

These synchronous versions exhibit a single-copy semantics: they behave as if the read or update were executed directly on the server.

In practice, we found that for most applications, the majority of reads and updates need not be strongly consistent. However, there often remain a few situations (e.g. finalizing a reservation, ending an auction, or joining a game with an upper limit on the number of players) where true arbitration is required, and where we are willing to pay the cost of synchronous communication (i.e. wait for the server to respond, or even block if offline). The ability of GSP to handle both synchronous and asynchronous reads and updates within the same framework is thus a major advantage.

Automatic Transactions. A prime scenario for GSP is the development of user-facing reactive event-driven applications, such as web applications or mobile apps. In that setting, we found it advantageous to automate the *push* and *pull* operations. Since the client program is already designed for cooperative concurrency and executes in event handlers, our framework can execute the following yield operation automatically between event handlers, and repeatedly when the event queue is empty:

```
yield() {
  push();
  pull();
}
```

All of the applications we wrote using the TouchDevelop platform rely on automatic transactions.

Comparison of Transactions. Our update transactions are different from conventional transactions (read-committed, serializable, snapshot isolation, or parallel snapshot isolation) since they do not check for any read or write conflicts. In particular, they never fail. The advantage is that they are highly available [2], i.e. progress is not hampered by network partitions. The disadvantage is that unlike serializable transactions (but like read-committed, snapshot, or parallel snapshot transactions), they not preserve all data invariants.

6 Robust Streaming

The GSP model described in the previous sections abstracts away many details that are important when we try to implement it in practice. In particular, it assumes that we have a RTOB implementation, it does not model failures of any kind, and it suffers from space explosion due to ever-growing update sequences.

In this section, we show that all of these issues are fixable without needing to change the abstract GSP protocol. Specifically, we describe a robust streaming server-client implementation of GSP. It explicitly models communication using sockets (duplex streams) and contains explicit transitions to model failures of the server, clients, and the network. Moreover, it eliminates all update sequences, and instead stores current server state and deltas in reduced form. Importantly, it is robust in the following sense:

- *Client programs never need to wait for operations to complete, regardless of failures in the network, server, or other clients.*
- Connections can fail at any time, on any end. New connections can replace failed ones and resume transmission correctly.

- The server may crash and recover, losing soft state in the process, but preserving persistent state. The persisted server state contains only a snapshot of the current state and the number of the last round committed by each client. It *does not store any logs*.
- Clients may crash silently or temporarily stop executing for an unbounded amount of time, yet are always able to reconnect. In particular, there are no timeouts (the protocol is fully asynchronous). Permanent failures of clients cannot disrupt the server, other clients, or violate the consistency guarantees.

Despite the more realistic communication and the possibility of channel and server failures, the streaming protocol remains faithful to the original protocol: we prove that it is a refinement, that is, all of its behaviors correspond to a behavior of the abstract protocol (transactional GSP, see Fig. 2). Thus, programmers may remain blissfully unaware of these complications.

6.1 States and Deltas

The streaming model does not store any update sequences (neither in the client, nor on the server). Instead, it eagerly reduces such sequences, and stores them in reduced form, either as *state objects* (if the sequence is a prefix of the global update sequence) or as *delta objects* (if the sequence is a segment of the global update sequence).

abstract type *State*
abstract type *Delta*

Deltas are produced by appending updates, or by reducing several deltas, and states are produced by applying deltas to the initial state:

const *initialstate*	:	*State*
function *read*	:	$Read \times State \rightarrow Value$
function *apply*	:	$State \times Delta^* \rightharpoonup State$
const *emptydelta*	:	*Delta*
function *append*	:	$Delta \times Update \rightharpoonup Delta$
function *reduce*	:	$Delta^* \rightharpoonup Delta$

Note that we define some of these functions as partial, reflecting that some updates may be invalid (in the cloud types model, these include incorrectly typed field updates or creation of a row with a duplicate unique identifier, for example).

▶ **Example 2.** For the key-value store, the implementation of this abstract interface is straightforward. We can represent both *State* and *Delta* as maps from keys to values, and define *reduce*, *append* and *apply* to simply merge such mappings (where the last write wins).

Correctness. Intuitively, a state-and-delta implementation correctly represents a given data model if the result of reading a state s by means of calling $read(r,s)$ yields the same result as reading $rvalue(r,u_1 \cdot u_n)$ where $u_1 \cdot u_n$ is the combined sequence of updates that led to the state s. More formally, we can define a overloaded *representation* relation $\lhd$ that relates state and delta objects to the update sequences they represent, as follows:

- On $Delta \times Update^*$, let $\lhd$ be the smallest relation such that (1) $emptydelta \lhd []$, and (2) $d \lhd a$ implies $append(d,u) \lhd a \cdot u$ for all updates u, and (3) $d_1 \lhd a_1 \wedge \cdots \wedge d_n \lhd a_n$ implies $reduce(d_1 \cdots d_n) \lhd a_1 \cdots a_n$.
- On $State \times Update^*$, let $\lhd$ be the smallest relation such that (1) $initialstate \lhd []$, and (2) $s \lhd a \wedge d_1 \lhd a_1 \wedge \cdots \wedge d_n \lhd a_n$ implies $apply(s,d_1 \cdots d_n) \lhd a \cdot a_1 \cdots a_n$.

```
class Channel {
  client : Client; // immutable
  Channel(c : Client) { client := c; }

  // duplex streams
  clientstream : Round * := [];                          // client to server
  serverstream : (GSPrefix | GSSegment) * := []; // server to client

  // server-side connection state
  accepted : boolean := false; // whether server has accepted connection

  // client-side connection state
  receivebuffer : (GSPrefix | GSSegment) *; // locally buffered packets
  established : boolean := false;              // whether client processed 1st packet
}
```

Figure 3 Channel Objects.

Now, we define a state-and-delta implementation to be *correct* if and only if $s \lhd a$ implies $read(r,s) = rvalue(r,a)$ for all reads r, states s and update sequences a.

Optimality. Subtleties arises when we care about space leaks. For the key-value store, for example, a sloppy implementation of the state object may fail to remove a key whose value is set to undefined from the map. We call such an implementation *non-optimal*, because some states occupy more space than needed (there exists a smaller representation of the same update sequence that is indistinguishable by queries).

Engineering state and delta objects to be optimal can be quite challenging once richer data types are considered, for example regarding dynamic creation and deletion of table rows. In the extended technical report [10, 11] we show an example of such a nontrivial optimal implementation of state and delta objects for the cloud types data model.

6.2 Channels

We model the network and communication sockets using *Channel* objects (Fig. 3). Channels contain two streams, one for each direction. We model streams using sequences, by adding elements on the right and removing them on the left. Channel objects also contain server-side and client-side connection state that can be read and written only on the respective side (i.e. it is not used for communication).

The sequence *clientstream* contains the rounds that the client sends to the server. The *Round* objects are defined as in GSP, except that they contain a delta object in place of a sequence of updates:

```
struct Round { origin : Client; number : ℕ; delta : Delta; }
```

The sequence *serverstream* contains reduced segments of the global sequence that the server streams to the client. When sending on a channel, a server always starts with a *GSPrefix* object (containing a *State* object), and then keeps sending *GSSegment* objects (containing *Delta* objects).

```
class GSPrefix { // represents a prefix of the global update sequence
  state: State := initialstate;
  maxround: (Client ⇀ ℕ) := {};
  method apply(s: GSSegment) {
    foreach((c,r) in s.maxround) { maxround[c] := r; }
    state := apply(state, s.delta);
} }
class GSSegment { // represents an interval of the global update sequence
  delta: Delta := emptydelta;
  maxround: (Client ⇀ ℕ) := {};
  method append(r: Round): void {
    maxround[r.origin] := r.number;
    delta := reduce(delta · r.delta);
} }
```

The method *apply* extends a prefix with a segment, and the method *append* extends a segment with a round. Both *GSPrefix* and *GSSegment* contain a partial map *maxround* that records the maximal client round of each client that is contained in the segment. Thus a client c receiving a prefix or segment can look at *maxround*[c] to determine the latest confirmed round.

6.3 Server

(Fig. 4) The server state is separated into persistent state (*serverstate*), which stores the current state and the number of the last round of each client that has been incorporated into the state, and soft state (*connections*) which stores currently active connections. A connection is started by the *accept_connection* transition, which adds it to the active connections *connections* and sets the *accepted* flag. It then sends the current state (i.e. the reduced prefix of the global sequence) on the channel.

During normal operation, the server repeatedly performs the *processbatch* operation. It combines a nondeterministic number of rounds (we use the * in the pseudocode to denote a nondeterministic choice) from each active connection into a single segment. This segment stores all updates in reduced form as a delta object. We then append this segment to the persistent state (which applies the delta to the current state, and updates the maximum round number per client), and send it out on all active channels.

The transition *drop_connection* models the abrupt failure or disconnection of a channel at the server side – but not on the client, who may still send and receive packets until it in turn drops the connection. Note that a client may reconnect later, using a fresh channel object, and will resend rounds that were lost in transit when the channel was dropped.

The transition *crash_and_recover* models a failure and recovery of the server, which loses all soft state but preserves the persistent state.

6.4 Client

(Fig. 5, 6) The fields of the client are similar to the transactional GSP client (Fig. 2), but with the following differences:

1. We use *State* and *Delta* objects instead of update sequences: *known* is now a *State* object, and *transactionbuf* is a *Delta* object.
2. There is a variable *channel* that contains the current connection (or null if there is none).

```
class StreamingServer {
  // persistent state
  serverstate: GSPrefix := new GSPrefix();
  // soft state
  connections: (Client ⇀ Channel) := {};
  // transitions
  accept_connection(ch: Channel) {
    requires ! ch.accepted && connections[ch.client] = null;
    ch.accepted := true;
    connections[ch.client] := ch;
    ch.serverstream := ch.serverstream · serverstate; // send first packet: current state
  }
  processbatch() {
    var s := new GSSegment();
    // collect updates from all incoming segments
    foreach((c,ch) in connections)
      receive(s, ch, *);
    // atomically commit changes to persistent state
    serverstate := serverstate.apply(s);
    // notify connected clients
    foreach((c,ch) in connections)
      ch.serverstream := ch.serverstream · s;
  }
  drop_connection(c: Client) { connections[c] := null; }
  crash_and_recover() { connections := {}; }
  // auxiliary functions
  receive(s: GSSegment, ch: Channel, count: int) {
    requires count <= ch.clientstream.length;
    foreach(r in ch.clientstream[0..count])
      s.append(r);
    ch.clientstream := ch.clientstream[count..];
  }
}
```

Figure 4 State and Transitions of the Streaming Server.

3. There is a new *pushbuffer*, which holds updates that were pushed but have not been sent on any channel yet (e.g. because there was no channel established at the time of the push). *rds_in_pushbuf* counts the number of rounds in the pushbuffer.
4. The *receivebuffer* is now stored inside the channel object.

The *read* transition computes the visible state by calling *curstate* which (nondestructively) applies the delta objects in *pending*, *pushbuf* and *transactionbuf* to the state object in *known*.

The *update* transition adds the given update to the transaction buffer, and clears the *tbuf_empty* flag (since delta objects do not have a function to query whether they are empty, we use a flag to determine whether the transaction buffer is empty).

The *confirmed* transition checks whether updates are pending in *pending* or *transactionbuf* or *pushbuffer*.

```
class StreamingClient {
  known : State := emptystate; // known prefix
  pending : Round * := []; // sent, but unconfirmed rounds
  round : ℕ := 0; // counts submitted rounds
  transactionbuf : Delta := emptydelta;
  tbuf_empty : boolean := true;
  channel : Channel := null;
  pushbuf : Delta := emptydelta; // updates that were pushed, but not sent yet
  rds_in_pushbuf : ℕ := 0; // counts the number of rounds in the pushbuffer
  // client interface transitions
  read(r : Read) : Value { return read(r, curstate()); }
  update(u : Update) {
    transactionbuf := transactionbuf.append(u);
    tbuf_empty := false;
  }
  confirmed() : boolean {
    return pending = [] && rds_in_pushbuf = 0 && tbuf_empty;
  }
  push() {
    pushbuf := pushbuf.append(transactionbuf);
    transactionbuf := []; tbuf_empty := true;
    rds_in_pushbuf := rds_in_pushbuf + 1; round := round + 1;
  }
  pull() {
    while (channel != null && channel.receivebuffer.length != 0) {
      var s := channel.receivebuffer[0];
      channel.receivebuffer := channel.receivebuffer[1..]
      if (channel.established) // not the first packet received on this channel
        assert(s instanceof GSSegment);
        known := known.append(s);
        adjust_pending_queue(s.maxround[this]);
      } else {
        channel.established := true;
        assert(s instanceof GSPrefix); // first packet contains latest server state
        known := s.state; // replace known prefix
        // resume sending rounds (remove confirmed, resend unconfirmed)
        adjust_pending_queue(s.maxround[this]);
        channel.clientstream := channel.clientstream · pending;
      }
    }
  }
// continued in next figure
```

Figure 5 States and transitions of the Streaming Client (part 1 of 2).

The *push* transition moves the content of the transactionbuffer to the pushbuffer, by combining and reducing the respective delta objects. The *pull* transition processes all packets in the receive buffer (which we model as part of the channel object), if any. When processing a packet, we track if this is the first packet received on this channel by checking the *established* flag.

- If the packet is not the first packet ($established = \mathsf{true}$), it is a GSSegment packet containing a delta object and representing an aggregation of one or more rounds. This delta object is then applied to *known*. Since the segment may also contain (reduced) echoes of one or more unconfirmed rounds, we determine the latest confirmed round $s.maxround[\mathsf{this}]$ and remove all rounds up to that one from the *pending* queue (in *adjust_pending_queue*).
- If the packet is the first packet, it is a GSPrefix packet containing the latest server state. We assign it to *known* and set *established* to true. However, we need to do some more work: since there may have been other channel objects used previously by this client, and dropped by the server at some point, we need to take care to resume the streaming of rounds with the correct round (to avoid losing or duplicating rounds). Since $s.maxround[\mathsf{this}]$ tells us the latest committed round on the server, we can ensure this by first removing confirmed rounds from the *pending* queue (using *adjust_pending_queue*), and then resending any rounds remaining in the *pending* queue.

The *receive* transition straightforwardly moves a packet from the serverstream into the receive buffer. The *drop_connection* transition models loss of connection at the client side. It removes the channel object from the client – but not from the server, who may still send and receive packets on the channel until it in turn drops the connection. The *send* transition requires that there is an established channel (this is is important to handle channel recovery correctly, as explained earlier). It sends one or more rounds (stored in *pushbuf*) as a single cumulative round with the latest pushed round number, and appends it to the *pending* queue. Note that in practice, we found it sensible to add additional preconditions on *send*, to limit the number of rounds in the pending buffer, and to avoid overflowing buffers in the network layer.

6.5 Refinement Proof

We now prove that the streaming client-server protocol (Fig. 3, 4, 5, 6) is a correct implementation, or a *refinement*, of the transactional GSP protocol (Fig. 2). This means that programmers need not worry about the intricacies of the former, but can safely assume that they are writing code for the latter: in particular, channel and server failures remain hidden beneath the protocol abstraction.

We define the set T_E of interface events to contain all expressions

$$read(c,r)(v) \mid update(c,u) \mid confirmed(c)(v) \mid push(c) \mid pull(c)$$

These events represent calls by the client program happening on some client c, and possibly returning a value v. The read and update events take a read operation r or an update operation u as a parameter.

We can now define a *trace* to be a finite or infinite sequence of interface events, and say a protocol *Impl* refines a protocol *Spec* if all traces of *Impl* are also traces of *Spec*. Since the events in the traces capture all interactions between the client program and the storage subsystem, refinement in this sense implies that if we run client programs on protocol *Impl*,

```
// class Client (continued)
  // auxiliary functions
  function curstate() : State {
    return apply(known, deltas(pending) · pushbuf · transactionbuf);
  }
  function deltas(seq : Round *) : Delta { return seq[0].delta · · · seq[seq.length - 1].delta; }
  procedure adjust_pending_queue(upto : ℕ) {
    while (upto >= pending[0].round) { pending := pending[1..]; }
  }
  // network transitions (nondeterministic)
  receive() {
    requires channel != null && channel.serverstream.length > 0;
    channel.receivebuffer := channel.receivebuffer · channel.serverstream[0];
    channel.serverstream := channel.serverstream[1..];
  }
  drop_connection() { channel := null; }
  send_connection_request() {
    requires channel == null;
    channel := new Channel(this);
  }
  send() {
    requires channel != null && channel.established && rds_in_pushbuf > 0;
    var r := new Round { origin = this, round = round - 1, delta = pushbuf };
    pending := pending · r;
    channel.clientstream := channel.clientstream · r;
    pushbuf := [];
    rds_in_pushbuf := 0;
  }
}
```

Figure 6 States and transitions of the Streaming Client (part 2 of 2).

they cannot detect a difference, i.e. all observable outcomes are consistent with running on protocol *Spec*.

For our proof, we use standard refinement proof methodology. We formalize a protocol as a labeled transition system $(\Sigma, \sigma^i, T, \delta)$ where Σ is a set of states, $\sigma^i \in \Sigma$ is the initial state, T is a set of transition labels, and $\delta \subset \Sigma \times T \times \Sigma$ is a transition relation. We write $\langle \sigma, t, \sigma' \rangle$ to represent an element of δ, that is, a transition with label t from state σ to state σ', and write $\langle \sigma_0, t_1, \sigma_1, t_2, \ldots, t_n, \sigma_n \rangle$ for a sequence of transitions with labels $t_1, \ldots, t_n$ (note that for $n = 0$, this is an empty transition sequence containing just a single state $\langle t_1 \rangle$).

We define transition systems for the implementation and specification to be $(\Sigma_I, \sigma_I^i, T_E \cup T_I, \delta_I)$ and $(\Sigma_S, \sigma_S^i, T_E \cup T_S, \delta_S)$, respectively, where the sets T_I, T_E, T_S represent categories of transitions, as follows.

We distinguish between *external* transitions T_E, which are exactly the interface events we have already defined above, and *internal* transitions T_S and T_I of the specification and implementation, respectively. Internal transitions usually represent nondeterministic events such as sending, receiving, or processing of messages that are not visible to the client program.

We define the set T_S of internal transitions of the specification to contain all expressions

$$\mathit{onreceive}(c,r) \mid \mathit{process}(r)$$

where c ranges over clients and r ranges over rounds (the rounds data structure). The *onreceive* transition is the handler for receiving an RTOB message in Fig. 2. The *process* transition represents the RTOB commit, i.e. the moment where a round becomes ordered into the global sequence. It is not explicitly listed in Fig. 2, since the RTOB is described abstractly there.

Naturally, the implementation has many more internal transitions than the specification, since it has more "moving parts". We define the set T_I of internal transitions of the implementation as

$$\begin{aligned} & \mathit{receive}(c) \mid \mathit{send}(c) \mid \mathit{processbatch}() \mid \mathit{crash_and_recover}() \\ \mid\ & \mathit{client_drop_connection}(c) \mid \mathit{server_drop_connection}(ch) \\ \mid\ & \mathit{accept_connection}(ch) \mid \mathit{send_connection_request}(c) \end{aligned}$$

where c ranges over clients and ch ranges over channel objects. All of these correspond to procedures with the same name in the code (Fig. 4,5,6).

We would like to emphasize that although we need to carefully consider all of these internal implementation transitions when proving refinement, the end result is that the programmer can be blissfully unaware of them.

6.5.1 Proof structure

To prove refinement, we construct an extended simulation relation
$R \subseteq \Sigma_I \times \Sigma_S \times \Sigma_A$,
where Σ_I is the implementation state, Σ_S is the specification state, and Σ_A is auxiliary state.

In our case, Σ_A is only needed to record history variables (we do not need prophecy variables as introduced in [1]). This means that Σ_A is updated according to some update function u_A that defines an auxiliary state $u_A((\sigma_I, \sigma_S, \sigma_A), t, \sigma_S)$ for each triple $(\sigma_I, \sigma_S, \sigma_A)$ and each transition $\langle \sigma_I, t, \sigma'_I \rangle$.

The following conditions capture the requirements on R to be a simulation. It is easy to see that if such an R exists, it implies trace refinement as desired.

1. R contains the initial state $\sigma^i = (\sigma^i_I, \sigma^i_S, \sigma^i_A)$
2. for all tuples $(\sigma_I, \sigma_S, \sigma_A) \in R$ and implementation transitions $\langle \sigma_I, t, \sigma'_I \rangle$, there exists a specification transition sequence $\langle \sigma = \sigma^0_S, t_1, \sigma^1_S, t_2, \ldots, t_n, \sigma^n_S \rangle$ (where $n \geq 0$) satisfying the following conditions:
   a. If $t \in T_I$ (that is, t is an externally unobservable transition of the implementation), then all the labels $t_1, \ldots, t_n$ must be in T_S (i.e. must be internal transitions of the specification).
   b. If $t \in T_E$ (that is, t is an externally observable transition of the implementation), then there must exist an i such that $t = t_i$ (one specification transition must match), and $t_j \in T_S$ for $j \neq i$ (the other specification transitions must be externally unobservable).
   c. When we move all three state components forward, that is, when we (1) apply the implementation transition to the implementation state, (2) apply the specification transition sequence to the specification state, and (3) update the auxiliary state, we stay in the relation: $(\sigma'_I, \sigma^n_S, u_A((\sigma_I, \sigma_S, \sigma_A), t, \sigma_S)) \in R$.

To define this relation and prove the obligations, we construct a "combined transition system" in the extended technical report [11] whose state is $\Sigma = \Sigma_I \times \Sigma_S \times \Sigma_A$, and which has the transitions described in (2.c) above. The simulation relation R captures the relation between rounds, clients, global sequences, and channels in the specification and the implementation. We can think of R as an *invariant* of this combined transition system, rather than a simulation relation. This helps to organize the proof in a familiar way, by stating invariants and transitions, and proving preservation.

6.6 Further optimizations

In our prototype we implemented a few additional optimizations left out here for simplicity. They include:

- The server caches recent deltas. When clients reconnect, and the server still has the relevant deltas in its cache, the server sends only the deltas needed instead of the whole state.
- The server, when sending segments to a client c, includes not the whole $maxround$, but only $maxround[c]$.
- As written, reads are potentially inefficient, thus some optimizations may be required. For example, in our implementation of the cloud types model, we store updates of fields inside objects representing the fields, and we cache the result of expensive reads, such as table enumerations.

The implementation presented here uses a single server, which is appropriate for modest read/write loads. The server can be easily made reliable, even on unreliable cloud compute infrastructure, by using reliable cloud storage to store the persistent state. Devising implementations that scale to heavier loads, while certainly possible, is beyond the scope of this paper.

7 Implementation in TouchDevelop

We have realized the ideas presented in this paper and their implementation as an extension of the web-based IDE and runtime system called TouchDevelop [30]. We implemented the streaming model using a Azure cloud service backed by Azure table storage (for the persistent state). Clients are written in JavaScript, run in webbrowsers, persist the local data in HTML5 local storage (and thus remain available offline), and connect to the service using websockets.

Rather than a basic key-value store data model, the TouchDevelop language supports full *cloud types*Z [5, 14] which include tables, indices, and records. We describe the cloud types data model and prove optimal reduction in the extended technical report [11].

For illustration, we give a few examples of TouchDevelop apps that use cloud types below: feel free to run them, inspect and edit their code, and create your own variations! The first two examples below have been contributed by our user community. In all of these examples, the use of cloud types is very simple, with the exception of the Cloud Game Selector which involves tricky synchronization and a flush operation.

- **Relatd** [sic] (http://tdev.ly/ruef) Lets users enter their qualities (either from a list, or freely entered) and finds and displays other users that share them.
- **Chatter Box** (http://tdev.ly/spji) A chat application.
- **TouchDevelop Jr.** (http://tdev.ly/vkrpa) Program a tiny robot using a simple language, then share your scripts with other users.

- **Instant Poll** (http://tdev.ly/nggfa) An app for quickly polling an audience and displaying the responses as a grid of colors.
- **Expense Recorder** (http://tdev.ly/nvoha) Allows easy recording of expenses in a table.
- **Contest Voting** (http://tdev.ly/etww) Used to determine the winner of the "Touch of Summer" coding contest.
- **Cloud List** (http://tdev.ly/blqz) A general-purpose list that can be concurrently edited.
- **Cloud Game Selector** (http://tdev.ly/nvjh) A library for matching multiple players to play games together.
- **Cloud Paper Scissors** (http://tdev.ly/sxjua) A simple rock/paper/scissors game that uses the cloud game selector library.

Other researchers have also experimented with refactoring non-cloud data in TD scripts into cloud types. A formative study shows that refactoring is applicable, relevant, and saves human effort [18].

8 Related Work

Eventual Consistency is motivated by the impossibility of achieving strong consistency, availability, and partition tolerance at the same time, as stated by the CAP theorem [17]. EC across the literature uses a variety of techniques to propagate updates (e.g. general causally-ordered broadcast [24, 26]). For a general high-level comparison of eventual consistency notions appearing in the literature, see [3, 6, 8].

Bayou's weakly consistent replication [29] follows a similar overall system design. However, it does not articulate an abstract reference model like GSP, or a data model. Conflicts are not resolved simply by ordering updates, but require explicit merge functions provided by the user.

As explained earlier, our Global Sequence Protocol (GSP) is an adaptation of a reliable total order broadcast [12, 16]. However, we go beyond prior work on broadcast, as we articulate the concept of a data model, describe how to reduce updates, and discuss optimality of reduction.

A version of core GSP supporting arbitrary replicated data types appears in [8], but without synchronization, transactions, or a robust implementation.

As explained in Section 4.3, GSP is similar, but not equivalent to the TSO memory model. In particular, GSP allows data to be read and updated offline without requiring communication. We could call GSP "the TSO for distributed systems". Neither TSO, nor any other memory consistency model we know of, allows arbitrary data models like GSP, or provides transactional synchronization via *push* and *pull*.

Just like our work, replicated data types and in particular CRDTs [7, 24, 24] provide optimized distributed protocols for certain data types. However, CRDTs are not easy to customize and compose, since the consistency protocol is not cleanly separated from the data model as in GSP, but specialized for a particular, fixed data type.

As explained earlier, the original work on cloud types [5, 14], while providing a comprehensive, composable way to define replicated structured data, falls short of providing either a simple reference model or a robust implementation.

The Jupiter system [22] has a similar system structure (client-server with bidirectional streaming) as our streaming model. However, it uses an *operational transformation* (OT) algorithm to transform conflicting updates with respect to each other, instead of simply ordering updates sequentially as in GSP.

The OT approach [13, 25, 27] provides an interesting and powerful (but arguably also somewhat confusing and error-prone [20]) conflict resolution mechanism that has seen successful application and even industrial adoption for collaborative editing applications. However, it comes at the expense of scalability. OT transformations grow quadratically with the number of concurrent updates, and prevent extended offline operation since that requires storing and later processing of update sequences.

In our situation, all the example applications used structured data that was entirely expressible using cloud types, which by design avoid the need for OT. Thus, we were not inclined to pay the price for providing operational transformations as part of our data model (but may revise this choice in the future).

9 Conclusion

We have motivated, defined, and explained the global sequence protocol (GSP), a simple operational reference model for replicated, eventually consistent shared data. We then showed how to implement it, presenting a robust streaming implementation that can hide network, server, and client failures, and reduces update sequences, while conforming to the GSP reference model.

Both GSP and the streaming implementation are parameterized by an abstract data model, and thus apply to a wide range of data types from simple key-value stores to the full cloud types model.

We hope that our work provides a path for simpler development of distributed applications on mobile devices. In the future, we would like to further investigate the layering of data abstractions and how to best support user-defined data models. We are also considering more work on the refinement proof, such as obtaining a mechanically verified proof, and/or adding fairness and liveness. Finally, we are working on extending GSP to partial replication scenarios.

Acknowledgments. We would like to thank Alexey Gotsman for interesting discussions on the GSP consistency model, and Adam Morrison for pointing out a mistake we made in our reasoning when comparing GSP to TSO in an earlier version of this paper.

References

1 M.Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2), 1991.
2 P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. In *International Conference on Very Large Databases (VLDB)*, 2014.
3 P. Bernstein and S. Das. Rethinking eventual consistency. In *SIGMOD International Conference on Management of Data*, SIGMOD'13, pages 923–928. ACM, 2013.
4 Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC'00*, 2000.
5 S. Burckhardt, M. Fähndrich, D. Leijen, and B. Wood. Cloud types for eventual consistency. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 7313 of *LNCS*, pages 283–307. Springer, 2012.
6 S. Burckhardt, A. Gotsman, and H. Yang. Understanding eventual consistency. Technical Report MSR-TR-2013-39, Microsoft, 2013.
7 S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *Principles of Programming Languages (POPL)*, 2014.

8 Sebastian Burckhardt. Principles of eventual consistency. *Foundations and Trends in Programming Languages*, 1(1-2):1–150, 2014.

9 Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Mooly Sagiv. Eventually Consistent Transactions. In *European Symposium on Programming (ESOP), (extended version available as Microsoft Tech Report MSR-TR-2011-117), LNCS*, volume 7211, pages 64–83, 2012.

10 Sebastian Burckhardt, Daan Leijen, and Manuel Fähndrich. Cloud types: Robust abstractions for replicated shared state. Technical Report MSR-TR-2014-43, Microsoft Research, March 2014.

11 Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. Technical Report MSR-TR-2015-11, Microsoft Research, April 2015. Extended version.

12 Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

13 M. Cart and J. Ferrie. Asynchronous reconciliation based on operational transformation for p2p collaborative environments. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2007)*, pages 127–138, Nov 2007.

14 Tim Coppieters, Laure Philips, Wolfgang De Meuter, and Tom Van Cutsem. An open implementation of cloud types for the web. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC'14, pages 2:1–2:2, New York, NY, USA, 2014. ACM.

15 G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Symposium on Operating Systems Principles*, pages 205–220, 2007.

16 X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

17 S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

18 M. Hilton, A. Christi, D. Dig, M. Moskal, S. Burckhardt, and N. Tillmann. Refactoring local to cloud data types for mobile apps. In *MobileSoft'14*. ACM, 2014.

19 IEEE Computer. CAP retrospective edition. *IEEE Computer*, 45(2), 2012.

20 A. Imine, M. Rusinowitch, G. Oster, and P. Molli. Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science*, 351:167–183, 2006.

21 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In SOSP'11, 2011.

22 D. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *User interface and software technology (UIST)*, 1995.

23 M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report Rapport de recherche 7506, INRIA, 2011.

24 Mark Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *13th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, Grenoble, France, October 2011.

25 M. Suleiman, M. Cart, and J. Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Conference on Supporting Group Work*, GROUP '97, pages 435–445. ACM, 1997.

26 C. Sun and C. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Computer Supported Cooperative Work*, CSCW'98, pages 59–68. ACM, 1998.

27 D. Sun and C. Sun. Operation context and context-based operational transformation. In *Conference on Computer Supported Cooperative Work*, CSCW'06, pages 279–288. ACM, 2006.

28 D. Terry, A. Demers, K. Petersen, M. Spreitzer M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

29 D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29:172–182, December 1995.

30 N. Tillmann, M. Moskal, J. de Halleux, and M. Fähndrich. Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In *ONWARD'11 at SPLASH (also available as Microsoft TechReport MSR-TR-2011-49)*, 2011.

31 D. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.

Streams à la carte: Extensible Pipelines with Object Algebras

Aggelos Biboudis[1], Nick Palladinos[2], George Fourtounis[1], and Yannis Smaragdakis[1]

1 **Dept. of Informatics and Telecommunications**
University of Athens, Greece
`{biboudis,gfour,smaragd}@di.uoa.gr`

2 **Nessos Information Technologies S.A.**
Athens, Greece
`npal@nessos.gr`

Abstract

Streaming libraries have become ubiquitous in object-oriented languages, with recent offerings in Java, C#, and Scala. All such libraries, however, suffer in terms of extensibility: there is no way to change the semantics of a streaming pipeline (e.g., to fuse filter operators, to perform computations lazily, to log operations) without changes to the library code. Furthermore, in some languages it is not even possible to add new operators (e.g., a `zip` operator, in addition to the standard `map`, `filter`, etc.) without changing the library.

We address such extensibility shortcomings with a new design for streaming libraries. The architecture underlying this design borrows heavily from Oliveira and Cook's object algebra solution to the expression problem, extended with a design that exposes the push/pull character of the iteration, and an encoding of higher-kinded polymorphism. We apply our design to Java and show that the addition of full extensibility is accompanied by high performance, matching or exceeding that of the original, highly-optimized Java streams library.

1998 ACM Subject Classification D.2.2. Software libraries, D.1.5 Object-oriented Programming, D.3.3 Language Constructs and Features

Keywords and phrases object algebras; streams; extensibility; domain-specific languages; expression problem; library design

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.591

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.1.1.9`

1 Introduction

Recent years have seen the introduction of declarative streaming libraries in modern object-oriented languages, such as Java, C#, or Scala. Streaming APIs allow the high-level manipulation of value streams (with each language employing slightly different terminology) with functional-inspired operators, such as `filter`, or `map`. Such operators take user-defined functions as input, specified via local functions (lambdas). The Java example fragment below shows a "sum of even squares" computation, where the even numbers in a sequence are squared and summed. The input to `map` is a lambda, taking an argument and returning its square. Similarly, the input to `filter` is a lambda, determining whether its argument is even.

© Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis;
licensed under Creative Commons License CC-BY

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 591–613

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
int sum = IntStream.of(v)
  .filter(x -> x % 2 == 0)
  .map(x -> x * x)
  .sum();
```

Our work is based on the key observation that streaming operators introduce a separate (domain-specific) sub-language that is interpreted during program run-time. This observation is inspired by the architecture of the Java 8 streams library, which aggressively manipulates the streaming pipeline, as if the library calls were syntax nodes of an interpreted program. A pipeline of the form "`of(...).filter(...).map(...).sum()`" is formed with `sum` being at the outermost layer, i.e., right-to-left as far as surrounding code is concerned. However, when the terminal operator (`sum`) is reached, it starts evaluation over the stream data by eventually invoking an iteration method in operator `of`. It is this method that drives iteration and calls the operators left-to-right. The result of such manipulation is significant performance gains. The Java 8 streams implementation effectively changes external (*pull-style*) iteration into internal (*push-style*). Recent benchmarking studies [2] report that, with this change, the library avoids a number of indirect calls and allows much better downstream optimizations.

The problem with existing library designs is that there is no way to alter the semantics of a streaming pipeline without changing the library itself. This is detrimental to library extensibility. For instance, a user may want to extend the library in any of the ways below:

- Create push-vs-pull versions of all operators.
- Create a logging interpretation of a pipeline, which logs actions and some intermediate results.
- Create an interpretation computing asynchronous versions of an evaluation (futures-of-values instead of values).
- Create an optimizing interpretation that fuses together operators, such as neighboring `filter`s or `map`s.

Additionally, the current architecture of streaming libraries prevents the introduction of new operators, precisely because of the inflexible way that evaluation is performed. As discussed above, Java streams introduce push-style iteration by default. This approach would yield semantic differences from pull-style iteration if more operators, such as `zip`, were added to the library. Furthermore, in some languages the addition of new operators requires editing the library code or using advanced facilities: in Java such addition is only possible by changing the library itself, while in C# one needs to use extension methods, and in Scala one needs to use implicits.

In our work, we propose a new design and architecture for streaming libraries for Java-like languages, to maximize extensibility without sacrificing on any other axis. Our approach requires no language changes, and only leverages features found across all languages examined – i.e., standard parametric polymorphism (generics).

Underlying our architecture is the object algebra construction of Oliveira and Cook [13] and Oliveira et al. [14]. This is combined with a library design that dissociates the push or pull nature of iteration from the operators themselves, analogously to the recent "defunctionalization of push arrays" approach in the context of Haskell [22].

Based on this architecture, we have implemented an alternative stream library for Java[1]. In our library, the pipeline shown earlier gets inverted and parameterized by an `alg` object, which designates the intended semantics. For instance, a plain Java-streams-like evaluation would be written:

---

[1] `http://biboudis.github.io/streamalg`

```
PushFactory alg = new PushFactory();
int sum = Id.prj(
           alg.sum(
            alg.map(x -> x * x,
             alg.filter(x -> x % 2 == 0,
              alg.source(v))))).value;
```

(The `Id.prj` and `value` elements, above, are part of a standard pattern for simulating higher-kinded polymorphism with plain generics. They can be ignored for the purposes of understanding our architecture. We discuss the pattern in detail in Section 4.)

Although the above fragment is slightly longer than the original, its elements are highly stylized. The user can adapt the code to other pipelines with trivial effort, comparable to that of the original code fragment in Java 8 streams. Most importantly, if the user desired a different interpretation of the pipeline, the only necessary change is to the first line of the example. An interpretation that has pull semantics and fuses operators together only requires a new definition of `alg`:

```
FusedPullFactory alg = new FusedPullFactory();
... // same as earlier
```

Such new semantics can be defined externally to the library itself. Adding `FusedPullFactory` requires no changes to the original library code, allowing for semantics that the library designer had not foreseen.

This highly extensible design comes at no cost to performance. The new architecture introduces no extra indirection and does not prevent the JIT compiler from performing any optimization. This is remarkable, since current Java 8 streams are designed with performance in mind (cf. the earlier push-style semantics). As we show, our library matches or exceeds the performance of Java 8 streams.

Overall, our work makes the following contributions:

- We introduce a new design and architecture[2] for streaming libraries and argue for its benefits, in terms of extensibility and low adoption barrier (i.e., use of only standard language features), all without sacrificing performance.
- We demonstrate extensibility and provide several alternative semantics for streaming pipelines, all in an actual, publicly available implementation.
- We provide an example of the use of object algebras in a real-world, performance-critical setting.

2 Background

We next discuss streaming libraries in Java, Scala, C#, and F#. We also introduce *push/internal* vs. *pull/external* iteration, via reference to specific facilities in these libraries.

2.1 Java

Java is a relative newcomer among streaming facilities, yet features a library that has received a lot of engineering attention. We already saw examples of the Java API for streaming in the introduction. In terms of implementation, the Java library follows a scheme that is highly optimized and fairly unique among statically typed languages.

[2] We follow the textbook distinction that "design" refers to how elements are separated into modules, while "architecture" refers to components-and-connectors, i.e., the machinery determining how elements of the design are composed. Our work shows a new library design, albeit one that would not be possible without a different underlying architecture.

In the Java 8 declarative stream processing API, operators fall into two categories: intermediate (*always lazy* – e.g., map and filter) and terminal (which can produce a value or perform side-effects – e.g., sum and reduce). For concreteness, let us consider the pipeline below. The expression (serving as a running example in this section) calculates the sum of squares of all values in an array of doubles.

```
public double sumOfSquaresSeq(double[] v) {
  double sum = DoubleStream.of(v)
  .map(d -> d * d)
  .sum();
  return sum;
}
```

The code first creates a sequential, ordered `Stream` of `double`s from an array that holds all values. The calls `map` and `sum` are an intermediate and a terminal operation respectively. The `map` operation returns a `Stream` and it is lazy. It simply declares the transformation that will occur when the stream will be traversed. This transformation is a stateless operation and is declared using a lambda function. The `sum` operation needs all the stream processed up to this point, in order to produce a value; this operation is *eager* and it is effectively the same as reducing the stream with the lambda `(x,y) -> x+y`.

Implementation-wise, the (stateless or stateful) operations on a stream are represented by objects chained together sequentially. A terminal operation triggers the evaluation of the chain. In our example, *if no optimization were to take place*, the `sum` operator would retrieve data from the stream produced by `map`, with the latter being supplied the necessary lambda expression. This traversing of the elements of a stream is realized through the `Spliterator` interface. This interface offers an API for traversing and partitioning elements of a source. A key method in this interface is `forEachRemaining` with signature

```
  void forEachRemaining(Consumer<? super T> action);
```

Normally, for the general case of standard stream processing, the implementation of `forEachRemaining` will internally call methods `hasNext` and `next` to traverse a collection, as well as `accept` to apply an operation to the current element. Thus, three virtual calls per element will occur.

However, stream pipelines, such as the one in our example, can be optimized. For the array-based `Spliterator`, the `forEachRemaining` method performs an indexed-based, do-while loop. The entire traversal is then transformed: instead of `sum` requesting the next element from `map`, the pipeline operates in the inverse order: `map` pushes elements through the `accept` method of its downstream `Consumer` object, which implements the `sum` functionality. (A `Consumer` in Java is an operation that accepts an argument and returns no result.) In this way, the implementation eliminates two virtual calls per step of iteration and effectively uses internal (push-style) iteration, instead of external (pull-style). This also enables further optimizations by the JIT compiler, often resulting in fully fused code.

The following (simplified for exposition) snippet of code is taken from the `Spliterators.java` source file of the Java 8 library and demonstrates this special handling, where `a` holds the source array and `i` indexes over its length:

```
do { consumer.accept(a[i]); } while (++i < hi);
```

The push-style iteration can be seen in this code. Each of the operators applicable to a stream needs to support this inverted pattern by supplying an `accept` operation. That operation, in turn, will call `accept` on whichever `Consumer<T>` may be downstream. The consumer of a push stream will provide a consumer function that is instantiated into the

iteration block of the stream.[3]

The dual of a push stream is a *pull stream*. Every combinator of a pull stream will build an iterator that will propagate some effect (e.g., apply a function `f` if this combinator is `map`) to each next element. C#, F# and Scala implement deferred execution over pipelines (all described in their respective sections) as pull streams. Java, on the other hand, supports push streams by default. Java additionally provides pull capabilities through the `iterator` combinator – we shall see in Section 3 why this facility is not equivalent to full pull-style iteration functionality.

2.2 Scala

Scala is an object-functional programming language for the JVM. Scala has a rich object system offering traits and mixin composition. As a functional language, it has support for higher-order functions, pattern matching, algebraic data types, and more. Since version 2.8, Scala comes with a rich collections library offering a wide range of collection types, together with common functional combinators, such as `map`, `filter`, `flatMap`, etc. The most general streaming API for Scala is that for lazy transformations of collections, which also avoids the creation of intermediate, allocated results.

To achieve lazy processing, one has to use the `view` method on a collection. This method wraps a collection into a `SeqView`. The following example illustrates the use of `view` for performing such transformations lazily:

```
def sumOfSquareSeq (v : Array[Double]) : Double = {
  val sum : Double = v
  .view
  .map(d => d * d)
  .sum
  sum
}
```

Ultimately, `SeqView` extends `Iterable[A]`. `SeqView` acts as a factory for iterators. As an example, we can demonstrate the common `map` function by mapping the transformation function to the source's `Iterable` iterator:

```
def map[T, U](source: Iterable[T], f: T => U) = new Iterable[U] {
  def iterator = source.iterator map f
}
```

The `Iterator`'s `map` function can then be implemented by delegation to the source iterator:

```
def map[T, U](source: Iterator[T], f: T => U): Iterator[U] = new Iterator[U] {
  def hasNext = source.hasNext
  def next() = f(source.next())
}
```

Note that there are 3 virtual calls (`next`, `hasNext`, `f`) per element pointed by the iterator. This is standard pull-style iteration, as in the unoptimized Java case, discussed earlier. Each operator has to "request" elements from the one supplying its input, rather than having a push-style pattern, with the producer calling the consumer directly.

[3] Intuitively, in internal (push-style) iteration, there is no co-routining between the loop and the consumer. The latter is fully under the control-flow of the former. (The call consuming data returns – with none of its local data escaping – before the next data are produced.)

2.3 C#/F#

C# is a modern object-oriented programming language targeting the .NET framework. An important milestone for the language was the introduction of several features in C# 3.0 in order to enable a more functional style of programming. These new features, under the umbrella of LINQ [12, 11], can be summarized as support for lambda expressions and function closures, extension methods, anonymous types and special syntax for query comprehensions. All of these enable the creation of new functional-style APIs for the manipulation of collections.

F# is a modern .NET functional-first programming language based on OCaml, with support for object-oriented programming, based on the .NET object system.

In C# we can program data streams as fluent-style method calls:

```
nums.Select(x => x * x).Sum();
```

or with the equivalent query comprehension syntactic sugar:

```
(from x in nums
 select x * x).Sum();
```

In F#, stream manipulation can be expressed as a direct pipeline of various combinators.

```
nums  |> Seq.map (fun x -> x * x)
      |> Seq.sum
```

C# and F# have near-identical operational behaviors and both C# methods (`Select`, `Where`, etc.) and F# combinators (`Seq.map`, `Seq.filter`, etc.) operate on `IEnumerable<T>` objects. The `IEnumerable<T>` interface can be thought of as a factory for creating iterators, i.e., objects with `MoveNext` and `Current` methods. The lazy nature of the iterators allows the composition of an arbitrary number of operators without worrying about intermediate materialization of collections between each call. For instance, the `Select` method returns an `IEnumerable` object that produces the iterator below:

```
class SelectEnumerator<T, R> : IEnumerator<R> {
  private readonly IEnumerator<T> inner;
  private readonly Func<T, R> func;
  public SelectEnumerator(IEnumerator<T> inner,
                          Func<T, R> func) {
    this.inner = inner;
    this.func = func;
  }
  bool MoveNext() { return inner.MoveNext(); }
  R Current { get { return func(inner.Current); } }
}
```

`SelectEnumerator` implements the `IEnumerator<R>` interface and delegates the `MoveNext` and `Current` calls to the inner iterator. From a performance point of view, it is not difficult to see that there is virtual call indirection between the chained enumerators. We have 3 virtual calls (`MoveNext`, `Current`, `func`) per element per iterator. Iteration is similar to Scala or to the general, unoptimized Java iteration: it is an external (pull-style) iteration, with each consumer asking the producer for the next element.

3 Stream Algebras

We next describe our stream library architecture and its design elements, including separate push and pull semantics, enhanced interpretations of a pipeline, optimizations and more.

```
Stream<Long> s = Stream.iterate(0L, i -> i + 2);

Iterator<Long> iterator = Stream
  .of(v)
  .flatMap(x -> s.map(y -> x * y))
  .iterator();

iterator.hasNext();
```

Figure 1 Infinite streams and flatMap.

3.1 Motivation

The goal of our work is to offer extensible streaming libraries. The main axis of extensibility that is not well-supported in past designs is that of pluggable semantics. In existing streaming libraries there is no way to change the evaluation behavior of a pipeline so that it performs, e.g., lazy evaluation, augmented behavior (e.g., logging), operator fusing, etc. Currently, the semantics of a stream pipeline evaluation is hard-coded in the definition of operators supplied by the library. The user has no way to intervene.

The original motivation for our work was to decouple the pull- vs. pull-style iteration semantics from the library operators. As discussed in Section 2, Java 8 streams are push-style by default, while Scala, C#, and F# streams are pull-style. A recent approach in the context of Haskell [22] performs a similar decoupling of push- vs. pull-style semantics through defunctionalization of the interface, yet affords no other extensibility.

Although Java 8 streams allow some pull-style iteration, they do not support fully pluggable pull-style semantics. The current pull-style functionality is via the `iterator()` combinator. This combinator is a terminal operator and adapts a push-style pipeline into an iterator that can be used via the `hasNext`/`next` methods. This is subtly different from changing the semantics of an entire pipeline into pull-style iteration.

For instance, the `flatMap` combinator takes as input a function that produces streams, applies it to each element of a stream, and concatenates the resulting streams. In a true pull-style iteration, it is not a problem if any of the intermediate streams happen to be infinite (or merely large): their elements are consumed as needed. This is not the case when a Java 8 `flatMap` pipeline is made pull-style with a terminal `iterator` call. Figure 1 shows a simple example. Stream `s` is infinite: it starts with zero and its step function keeps adding 2 to the previous element. The `flatMap` application produces modified copies of the infinite stream `s`, with each element multiplied by those of a finite array, `v`. Evaluation does not end until an out-of-memory exception is raised.[4]

Our library design removes such issues, allowing pipelines with pluggable semantics. Although the separation of pull- and push-style semantics was our original motivation, it soon became evident that an extensible architecture offers a lot more options for semantic extensibility of a stream pipeline. We discuss next the new architecture and several semantic additions that it enables.

[4] This is a known issue, which we have discussed with Java 8 streams implementors, and does not seem to have an easy solution. The underlying cause is that the type signatures of operators (e.g., `of` or `flatMap`) encode termination conditions as return values from downstream operators. For `flatMap` to avoid confusing the conditions from its parameter stream (result of `map` in this example) and its downstream (`iterator` in the example) it needs to evaluate one more element of the parameter stream than strictly needed, and that element happens to be infinite in the example.

3.2 Stream as Multi-Sorted Algebras

Our extensible, pluggable-semantics design of the library is implemented using an architecture based on object algebras. Object algebras were introduced by Oliveira and Cook [13] as a solution to the *expression problem* [25]: the need to have fully extensible data abstraction while preserving the modularity of past code and maintaining type safety. The need for extensibility arises in two forms: adding new data variants and adding new operations. Intuitively, an object algebra is an interface that describes method signatures for creating syntax nodes (data variants). An implementation of the algebra offers semantics to such syntax nodes. Thus, new data variants (syntax nodes) are added by extending the algebra, while new operations (semantics) correspond to different implementations of the algebra.

We next present the elements of the object algebra approach directly in our streaming domain.

In our setting, the set of variants to extend are the different combinators: `map`, `take` (called `limit` in Java), `filter`, `flatMap`, etc. These are the different cases that a semantics of stream evaluation needs to handle. The "operations" on those variants declare the manipulation/transformation that will be employed for all produced data items. We will use the term "behavior" for such operations.

Our abstraction for streams is a multi-sorted algebra. The two sorts that can be evolved as a *family* are the type of the stream, which can hold some type of values, and the type of the value produced by terminal operations. The signature of the former is called `StreamAlg` while the latter is `ExecStreamAlg`. The `Exec*` prefix is used to denote that this is the algebra for the types that perform execution. The algebras are expressed as generic interfaces and classes implementing these interfaces are *factories*. In our multi-sorted algebra these two distinct parts are connected with the subtyping relation and classes that implement the two interfaces can evolve independently, to form various combinations.

Intermediate Combinators. Our base interface, `StreamAlg`, is shown below.

```
interface StreamAlg<C<_>> {
    <T>    C<T> source(T[] array);
    <T, R> C<R> map(Function<T, R> f, C<T> s);
    <T, R> C<R> flatMap(Function<T, C<R>> f, C<T> s);
    <T>    C<T> filter(Predicate<T> f, C<T> s);
}
```

As can be seen, `StreamAlg` is parameterized by a unary type constructor that we denote by the `C<_>` syntax. This is a device used for exposition. That is, for the purposes of our presentation we assume *type-constructor* polymorphism (a.k.a. higher-kinded polymorphism): the ability to be polymorphic on type constructors. This feature is not available in Java (although it is in, e.g., Scala).[5] In our actual implementation, type-constructor polymorphism is emulated via a standard stylized construction, which we explain in Section 4.

Every combinator of streams is also a constructor of the corresponding algebra; it returns (*creates*) values of the abstract set. Each constructor of the algebra creates a new intermediate node of the stream pipeline and, in addition to the value of the previous node (parameter `s`) that it will operate upon, it takes a *functional interface*. (A functional interface has exactly one abstract method and is the type of a lambda in Java 8.)

[5] The original object algebras work of Oliveira and Cook [13] did not require type-constructor polymorphism for its examples. Later work by Oliveira et al. [14] used type-constructor polymorphism in the context of Scala.

```
class PushFactory implements StreamAlg<Push> {
  public <T> Push<T> source(T[] array) {
    return k -> {
      for(int i=0 ; i < array.length ; i++){
        k.accept(array[i]);
      }
    };
  }

  public <T, R> Push<R> map(Function<T, R> mapper, Push<T> s) {
    return k -> s.invoke(i -> k.accept(mapper.apply(i)));
  }
}
```

Figure 2 Example of a PushFactory.

Terminal Combinators. The `ExecStreamAlg` interface describes terminal operators, which trigger execution/evaluation of the pipeline. These operators are also parametric. They can return a scalar value or a value of some container type (possibly parameterized by some other type). For instance, `count` can return `Long`, hence having blocking (synchronous) semantics, or it can return `Future<Long>`, to offer asynchronous execution.

```
interface ExecStreamAlg<E<_>, C<_>> extends StreamAlg<C> {
    <T> E<Long> count(C<T> s);
    <T> E<T>    reduce(T identity, BinaryOperator<T> acc, C<T> s);
}
```

Once again, this algebra is parameterized by unary type constructors and it also carries as a parameter the abstract stream type that it will pass to its super type, `StreamAlg`.

3.3 Adding New Behavior for Intermediate Combinators

We next discuss the extensibility that our design affords, with several examples of different interpretation semantics.

Push Factory. The first implementation in our library is that of a push-style interpretation of a streaming pipeline, yielding behavior equivalent to the default Java 8 stream library.

Push-style streams implement the `StreamAlg<Push>` interface (where `Push` is the *container* or *carrier type* of the algebra). All combinators return a value of some type `Push<...>`, i.e., a type expression derived from the concrete constructor `Push`. Our `PushFactory` implementation, restricted to combinators `source` and `map`, is shown below.

A `Push<...>` type is the embodiment of a push-style evaluation of a stream. It carries a function, which can be composed with others in a *push-y* manner. In the context of Java, we want to be able to assign lambdas to a `Push<...>` reference. Therefore we declare `Push<X>` as a functional interface, with a single method, `void invoke(Consumer<T>)`. The `Consumer<T>` argument is itself a lambda (with method name `accept`) that takes as a parameter an item of type `T` and returns void. This consumer can be thought of as the *continuation* of the evaluation (hence the conventional name, `k`). The entire stream is evaluated as a loop, as shown in the implementation of the `source` combinator, above. `source` returns a lambda that takes as a parameter a `Consumer<T>`, iterates over the elements of a source, `s`, and passes elements one-by-one to the consumer.

Similarly, the `map` operator returns a push-stream embodiment of type `Push<...>`. This stream takes as argument another stream, `s`, such as the one produced by `source`, and

invokes it, passing it as argument a lambda that represents the map semantics: it calls its continuation, k, with the argument (i.e., the element of the stream) as transformed by the mapping function. This pattern follows a similar continuation-passing-style convention as in the original Java 8 streams library. (As discussed in Section 2.1, this reversal of the pipeline flow enables significant VM optimizations and results in faster code.)

The next combinator, whichever it is, will consume the transformed elements of type R. The implementation of other combinators, such as filter and flatMap, follows a similar structure.

Pull Factory. As discussed earlier, Java 8 streams do not have a full pull-style iteration capability. They have to fully realize intermediate streams, since the pull semantics is implemented as a terminal combinator and only affects the external behavior of an entire pipeline. (As we will see in our experiments of Section 6, this is also a source of inefficiency in practice.) Therefore, the first semantic addition in our library is pull-style streams.

Pull-style streams implement the StreamAlg<Pull> interface. In this case Pull<T> is an interface that represents iterators, by extending the Iterator<T> interface. For pull semantics, each combinator returns an anonymous class – one that implements this interface by providing definitions for the hasNext and next methods. In Figure 3 we demonstrate the implementation of the source and map operators, which are representative of others.

We follow the Java semantics of iterators (the effect happens in hasNext). Each element that is returned by the next method of the map implementation is the transformed one, after applying the needed mapper lambda to each element that is retrieved. The retrieval is realized by referring to the s object, which carries the iterator of the previous pipeline step.

Note how dissimilar the Push and Pull interfaces are (a lambda vs. an iterator with next and hasNext). Our algebra, StreamAlg<C<_» is fully agnostic regarding C, i.e., whether it is Push or Pull.

Log Factory. With a pluggable semantics framework in place, we can offer several alternative interpretations of the same streaming pipeline. One such is a logging implementation. The log factory expresses a cross-cutting concern, one that interleaves logging capabilities with the actual execution of the pipeline. Although the functionality is simple, it is interesting in that it takes a mixin form: it can be merged with other semantics, such as push or pull factories. The code for the LogFactory, restricted to the map and count operators, is shown in Figure 4.

The code employs a delegation-based structure, one that combines an implementation of an execution algebra (of any behavior for intermediates and orthogonally of any behavior for terminal combinators) with a logger. We parameterize LogFactory with an ExecStreamAlg and then via delegation we pass the intercepted lambda as the mapping lambda of the internal algebra. For example, if the developer has authored a pipeline alg.reduce(0L, Long::sum, alg.map(x -> x + 2, alg.source(v))), then, instead of using an ExecPushFactory that will perform push-style streaming, she can pass a LogFactory<>(new ExecPushFactory()) effectively mixing a push factory with a log factory.

Fused Factory. An interpretation can also apply optimizations over a pipeline. The optimization is applied automatically, as long as the user chooses an evaluation semantics that enables it. This is effected with an extension of a PullAlgebra that performs fusion of adjacent operations. Using a FusedPullFactory the user can transparently enable fusion

```
class PullFactory implements StreamAlg<Pull> {
  public <T> Pull<T> source(T[] array) {
    return new Pull<T>() {
      final int size = array.length;
      int cursor = 0;
      public boolean hasNext() { return cursor != size; }
      public T next() {
        if (cursor >= size)
          throw new NoSuchElementException();
        return array[cursor++];
      }
    };
  }
  public <T, R> Pull<R> map(Function<T, R> mapper, Pull<T> s) {
    return new Pull<R>() {
      R next = null;
      public boolean hasNext() {
        while (s.hasNext()) {
          T current = s.next();
          next = mapper.apply(current);
          return true;
        }
        return false;
      }
      public R next() {
        if (next != null || this.hasNext()) {
          R temp = this.next;
          this.next = null;
          return temp;
        } else throw new NoSuchElementException();
      }
    };
  }
}
```

Figure 3 Example of PullFactory functionality.

for multiple `filter` and multiple `map` operations. In this factory, the two combinators are redefined and, instead of creating values of an anonymous class of type `Pull`, they create values of a refined version of the `Pull` type. This gives introspection capabilities to the `map` and `filter` operators. They can inspect the dynamic type of the stream that they are applied to. If they operate on a fusible version of `map` or on a fusible version of `filter` then they proceed with the creation of values for these extended types with the composed operators. We elide the definition of the factory, since it is lengthy.

3.4 Adding New Combinators

Our library design also allows adding new combinators without changing the library code. In case we want to add a new combinator, we first have to decide in which algebra it belongs. For instance, we have added a `take` combinator without disturbing the original algebra definitions. A `take` combinator has signature `C<T> take(int n)` so it clearly belongs in `StreamAlg`. We have to implement the operator for both push and pull streams, but we want to allow the possibility of using `take` with any `ExecStreamAlg`. Our approach again uses delegation, much like the `LogFactory`, shown earlier in Figure 4. We create a generic `TakeStreamAlg<E, C>` interface and orthogonally we create an interface `ExecTakeStreamAlg<E, C>` that extends `TakeStreamAlg<C>` and `ExecStreamAlg<E, C>`. In the case of push streams, `ExecPushWithTakeFactory<E>` implements the interface we created, where `C = Push`, by defining

```
class LogFactory<E<_>, C<_>> implements ExecStreamAlg<E, C> {
  ExecStreamAlg<E, C> alg;

  <T, R> C<R> map(Function<T, R> mapper, C<T> s) {
    return alg.map(i -> {
      System.out.print("map: " + i.toString());
      R result = mapper.apply(i);
      System.out.println(" -> " + result.toString());
      return result;
    }, s);
  }

  public <T> E<Long> count(C<T> s) {
    return alg.count(s);
  }
}
```

Figure 4 Example of LogFactory functionality.

the `take` operator. All other operators for the push case are inherited from the `PushFactory` supertype. The `ExecPushWithTakeFactory<E>` factory is parameterized by `ExecStreamAlg<E, Push> alg`. Generally, the factory can accept as parameter any algebra for terminal operators.

3.5 Adding New Behavior for Terminal Combinators

Future Factory. Our library design also enables adding new behavior for terminal combinators. The most interesting example in our current library components is that of `FutureFactory`: an interpretation of the pipeline that triggers an asynchronous computation. Instead of returning scalar values, a `FutureFactory` parameterizes `ExecStreamAlg` with a concrete type constructor, `Future<X>`.[6] (This is in much the same way as, e.g., a `PushFactory` parameterizes `StreamAlg` with type constructor `Push`, in Figure 2.) `Future` is a type that provides methods to start and cancel a computation, query the state of the computation, and retrieve its result.

`FutureFactory` defines terminal operators `count` and `reduce`, to return `Future<Long>` and `Future<T>` respectively. Intermediate combinators are defined similarly to the terminal ones, but are omitted from the listing.

4 Emulating Type-Constructor Polymorphism

As noted earlier, our presentation so far was in terms of type-constructor polymorphism, although this is not available in Java. For our implementation, we simulate type-constructor polymorphism via a common technique. The same encoding has been used in the implementation of object-oriented libraries – e.g., in type classes for Java [6] and in finally tagless interpreters for C# [10]. The technique was also recently presented formally by Yallop and White [27], and used to represent higher-kinded polymorphism in OCaml.

In this encoding, for an unknown type constructor `C<_>`, the application `C<T>` is represented as `App<t, T>`, where `T` is a Java class and `t` is a marker class that identifies the type constructor `C`. For example, our stream algebra shown in Section 3.2 is written in plain Java as follows:

[6] That is, `Future` is our own class, which extends the Java library class `FutureTask`, and not to be confused with the Java library `java.util.concurrent.Future` interface.

```
class ExecFutureFactory<C<_>> implements ExecStreamAlg<Future, C> {
  private final ExecStreamAlg<Id, C> execAlg;
  public <T> Future<Long> count(C<T> s) {
    Future<Long> future = new Future<>(() -> {
      return execAlg.count(s).value;
    });
    future.run();
    return future;
  }
  public <T> Future<T> reduce(T identity,
                              BinaryOperator<T> accumulator,
                              C<T> s) {
    Future<T> future = new Future<>(() -> {
      return execAlg.reduce(identity, accumulator, s).value;
    });
    future.run();
    return future;
  }
}
```

Figure 5 Count and reduce operators in FutureFactory.

```
public interface App<C, T> { }

public interface StreamAlg<C> {
    <T> App<C, T> source(T[] array);
    <T, R> App<C, R> map(Function<T, R> f, App<C, T> app);
    <T, R> App<C, R> flatMap(Function<T, App<C, R>> f, App<C, T> app);
    <T> App<C, T> filter(Predicate<T> f, App<C, T> app);
}
```

A subtle point arises in this encoding: given `C`, how is `t` generated? This class is called the "brand", as it tags the application so that it cannot be confused with applications of other type constructors; this brand should be extensible for new types that may be added later to the codebase. This means that (a) `t` should be a fresh class name, created when `C` is declared; and (b) there should be a protocol to ensure that the representation is used safely.

Brand freshness. The freshness of the brand name is addressed by declaring `t` as a nested class inside the class of the new type constructor. Since `t` exists at a unique point in the class hierarchy, no other class may declare a brand that clashes with it, and its declaration happens at the same time as `C` is declared. In the following, we see the encoding of the type constructor `Pull<T>`, with its `t` brand:

```
public interface Pull<T> extends App<Pull.t, T>, Iterator<T> {
    static class t { }
    static <A> Pull<A> prj(App<Pull.t, A> app) { return (Pull<A>) app; }
}
```

We see that the encoding above has an extra method `prj`, which does a downcast of its argument. The OCaml encoding of Yallop and White needs two methods `inj` and `prj` (for "inject" and "project") that cast between the concrete type and the instantiation of the type application. In Java, we define `prj`, which takes the representation of the type application and returns the actual `Push<T>` instantiation. In contrast to OCaml, Java has subtyping, so `inj` functions are not needed: a `Pull<T>` object can always be used as being of type `App<Pull.t, T>`. The `Iterator` interface in the declaration above is not related to the encoding, but is part of the semantics of pull-style streams.

```
Long result = Id.prj(alg.count(
                       alg.filter(x -> x % 2L == 0,
                         alg.source(v)))).value;
```

Figure 6 Count of filtered items.

```
<E, C> App<E, Long> cart(ExecStreamAlg<E, C> alg) {
  return alg.reduce(0L, Long::sum,
           alg.flatMap(
             x -> alg.map(y -> x * y,
                    alg.source(v2)),
             alg.source(v1)));
}
```

Figure 7 Sum of the cartesian product.

Safely using the encodings. This encoding technique has a single unchecked cast, in the `prj` function. The idea is that the cast will be safe if the only way to get a value of type `App<Pull.t, X>` (for any `X`) is if it is really a value of the subtype, `Pull<X>`. This property clearly holds if values of type `App<Pull.t, X>` (or values of any type involving `Pull.t`) are never constructed. In the Yallop and White technique for OCaml, this is ensured syntactically by the "freshness" of the brand, `t`, which is private to the type constructor. In Java, the property is ensured by convention: every subtype `S` of `App` has a locally defined brand `t` and no subtype of `App<S.t, X>` other than `S` exists.

Type expressions without type-constructor polymorphism. Another detail of the encoding is the representation of type expressions that are not parametric according to a type constructor; for those we need an identity type application, `Id`.

```
public class Id<T> implements App<Id.t, T> {
    public final T value;
    public Id(T value) { this.value = value; }
    public static class t { }
    public static <A> Id<A> prj(App<Id.t, A> app) { return (Id<A>) app; }
}
```

Using the class above, the type expression `List<Integer>` can then be represented as `Id<List<Integer»`.

5 Using Streams

With the encoding of type-constructor polymorphism, our description of the library features is complete. A user can employ all combinators to build pipelines, and can flexibly choose the semantics of these pipelines.

The example of Figure 6 declares a pipeline that filters long integers and then counts them. The expression assumes an implementation, `alg`, of a stream algebra. Note that the prefix, `Id.prj`, and suffix, `value`, of the pipeline expression are only needed for our type-constructor polymorphism simulation.

Similarly, Figure 7 constructs a sum of the cartesian product pipeline between two arrays. The factory object (implementing the algebras) is factored out and becomes a parameter of the method `cart`.

```
Id.prj(cart(new ExecPushFactory())).value;
Id.prj(cart(new ExecPullFactory())).value;
Id.prj(cart(new ExecFusedPullFactory())).value;
Id.prj(cart(new LogFactory<>(new ExecPushFactory()))).value;
Id.prj(cart(new LogFactory<>(new ExecPullFactory()))).value
Future.prj(cart(new ExecFutureFactory<>(new ExecPushFactory())));
Future.prj(cart(new ExecFutureFactory<>(new ExecPullFactory())));
```

Figure 8 Examples.

The above can be used with any of the various semantics factories presented in Section 3, depending on the kind of evaluation the user wants to perform. In Figure 8 we present a summary of all the combinations of factories that can be used. The first five expressions return a scalar value `Long` and the last two a `Future<Long>`.

6 Performance

It is interesting to assess the performance of our approach, compared to the highly optimized Java 8 streams. Since our techniques add an extra layer of abstraction, one may suspect they introduce inefficiency. However, there are excellent reasons why our design can yield high performance:

- Object algebras are used merely for pipeline construction and not for execution. Once the data processing loop starts, it should be as efficient as in standard Java streams.
- Our design offers fully pluggable semantics. This is advantageous for performance. We can leverage fusion of combinators, proper pull-style iteration without materialization of full intermediate results, and more.

Our benchmarks aim to showcase these two aspects. In this sense, some of the benchmarks are unfair to Java 8 streams: they explicitly target cases for which we can optimize better. We point out when this is the case.

We use a set of microbenchmarks offering various combinations of streaming pipelines:[7]

- **reduce**: a sum operation.
- **rfilter/reduce**: a filter-sum pipeline.
- **rfilter/map/reduce**: a filter-map-sum pipeline.
- **rcart/reduce**: a nested pipeline with a `flatMap` and an inner operation, with a `map` (capturing a variable), to encode the sum of a Cartesian product.
- **rfused filters**: 8 consecutive filter operations and a count terminal operation. The implementation is push-style for Java 8, push-style, pull-style & fused for our library.
- **fused maps**: 8 consecutive map operations and a count terminal operation. The implementation is push-style for Java 8, push-style, pull-style & fused for our library.
- **rcount**: a count operation (pull-style).
- **rfilter/count**: a filter-count pipeline (pull-style).

[7] The benchmark programs are adapted from our earlier benchmarking study [2] of streams in various languages. As shown in that study, Java 8 streams typically significantly outperform other implementations. Still, performance of streaming libraries lags behind hand-optimized code. This is to be expected, since hand-written code can remove most overhead of lazy evaluation, by fusing the consumer of data with the producer. JDK developers have shown significant interest in future VM optimizations that will allow Java streams to approach the performance of hand-written code [16].

- **rfilter/map/count**: a filter-map-count pipeline (pull-style).
- **rcart/take/count**: a nested pipeline with a `flatMap` and an inner operation, with a `map`, to encode taking the first few elements of a Cartesian product and then counting them (pull-style).

Although our library is not yet full-featured, it faithfully (relative to Java 8 streams) implements the facilities tested in these benchmarks.

Input: All tests were run with the same input set. For all benchmarks except **cart/reduce** and **cart/take/count** we used an array of $N = 10,000,000$ `Long` integers (boxed),[8] produced by N integers with a `range` function that fills the arrays procedurally. The **cart/reduce** test iterates over two arrays. An outer one of $10,000,000$ long integers and an inner one of 10. For the **cart/take/count** test, the sizes of the inner and outer arrays are reversed and the `take` operator draws only the first $n = 100,000$ elements. Fusion operations use $1,000,000$ long integers.

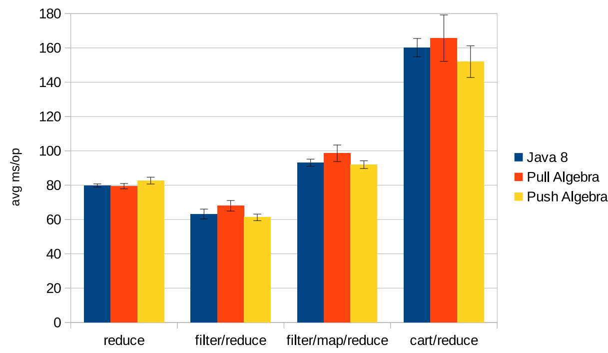
Setup: We use a Fedora Linux x64 operating system (version 3.17.4-200.fc20) that runs natively on an Intel Core i5-3360M vPro 2.8GHz CPU (2 physical x 2 logical cores). The total memory of the system is 4GB. We use version 1.8.0.25-4.b18 of the Open JDK.

Automation: We used the Java Microbenchmark Harness (JMH) [17]: a benchmarking tool for JVM-based languages that is part of the OpenJDK. JMH is an annotation-based tool and takes care of all intrinsic details of the execution process, in order to remove common experimental biases. The JVM performs JIT compilation so the benchmark author must measure execution time after a certain warm-up period to wait for transient responses to settle down. JMH offers an easy API to achieve that. In our benchmarks we employed 10 warm-up iterations and 10 proper iterations. We also force garbage collection before benchmark execution. Additionally, we used 2 VM-forks for all tests, to measure potential run-to-run variance. We have fixed the heap size to 3GB for the JVM to avoid heap resizing effects during execution.

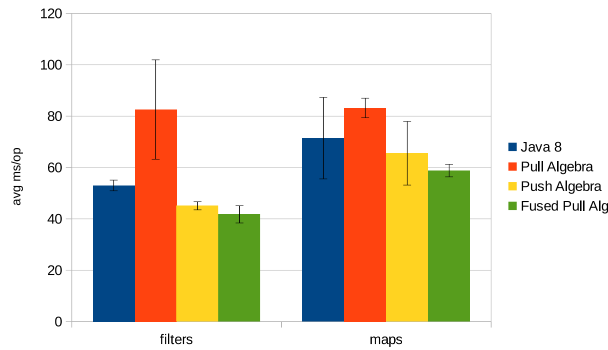
Results: The benchmarks are cleanly grouped in 4 sets:

- In Figure 9a we present the results of the first 4 benchmarks: **reduce**, **filter/reduce**, **filter/map/reduce**, and **cart/reduce**. These are "fair" comparisons, of completely equivalent functionality in the two libraries. As can be seen, the performance of our push algebra implementation matches or exceeds that of Java 8, validating the claim that our approach does not incur undue overheads.
- Figure 9b presents the results for the next two benchmarks: **fused filters** and **fused maps**. These benchmarks are intended to demonstrate the improvement from our fusing semantics. The Java 8 implementation compared is push-style. Still, our fused pull-style semantics yield a successful optimization, outperforming even the efficient, push-style iteration. Due to our design, this optimization is achieved modularly and independently of the rest of the stream implementation.
- Figure 9c includes the next 3 benchmarks: **count**, **filter/count**, and **filter/map/count**. These are benchmarks of semantically uneven implementations. Java 8 streams support

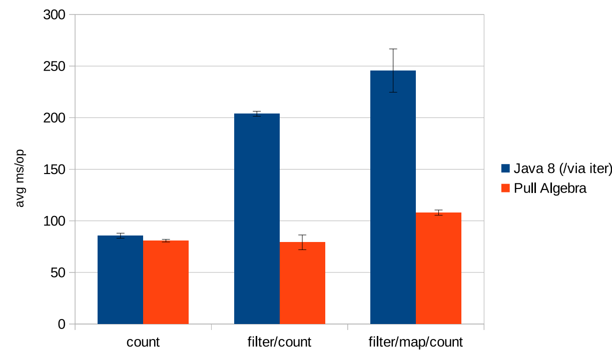
[8] Specialized pipelines for primitive types is not supported in our library, but should be a valuable future engineering addition.



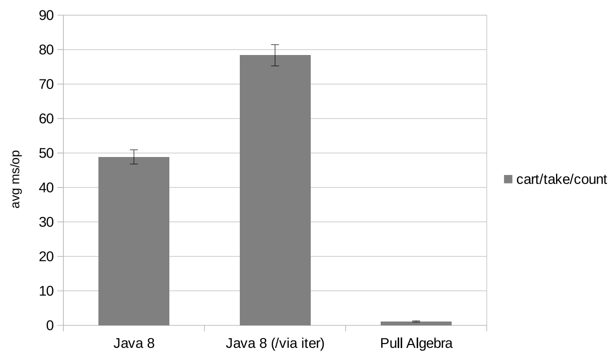
(a) Basic pipelines.



(b) Fusible pipelines.



(c) Pull based reduction.



(d) Pull & Push based flatMap/take.

Figure 9 Microbenchmarks on JVM in milliseconds / iteration (average of 10).

pull-style functionality by transforming the stream into an iterator, but this is not equivalent to full pull-style iteration. As can be seen, a true pull semantics for all operators can be much faster.

- Finally in Fig 9d we show the **cart/take/count** benchmark. This contains a pipeline that is pathological for the Java pull-style streams, in much the same way as the infinite evaluation of Section 3.1. (Push-style streams have the same pathology, by definition: they cannot exploit the fact that only a small number of results are needed in the final operator.) Instead of an infinite stream we reproduce the nested `flatMap`/`map` pipeline but with the larger array being the nested one. As `flatMap` needs to materialize nested arrays (effectively applying the inner `map` function to create the inner stream) it suffers from the effect of inner allocations. Our proper pull-style pipeline does not present this behavior. The result is spectacular in favor of our pull algebra implementation, because of the small number of elements actually needed by the `take` operator.

7 Discussion

We next present observations related to our library design and its constituent elements.

7.1 Fluent API

Object Algebras drive the "interpretation" of streams in our work, so a nested, reversed pattern occurs when declaring the combinators of a pipeline: instead of "`of(...).filter(...).count()`", our pipeline looks like "`alg.count(alg.filter(..., alg.source(...)))`". This reversed pattern follows the declaration order of the combinators, contradicting the natural ordering of a fluent API.

We created an experimental fluent API in Java using static methods in the interface of the object algebra,[9] but the result was cumbersome. In contrast, we created skeletal C# and Scala implementations of a fluent API for our library design, for demonstration purposes. (Our object algebra streaming architecture applies to any modern OO language with generics, so C# and Scala libraries based on the same principles can be developed in the future.)

In C# the user can create a fluent API through the use of extension methods. Extension methods enable the user to "add" methods to *existing types* without creating a new derived type, recompiling, or modifying the original type. Extension methods are simply a compiler shorthand that enables static methods to be called with instance syntax. Using that feature, the user can create a static class enclosing extension methods that capture the reversed flow.

Figure 10 shows the relevant C# code snippet. Extension methods are defined for the function type, `Func<F, App<C, T>>`. The user wraps all methods of an algebra with methods that, instead of returning `C<T>`, return a function that takes as parameter an algebra object. The algebra object is, thus, hidden from the original code and introduced implicitly in calls to such returned functions.

The above technique enables fluent ordering, as shown in the `Example` method of the listing. The fluent API also has immediate side benefits in current programming tools: the user is able to retrieve the list of combinators via the intelligent code completion feature of the IDE.

We also retrieve fluency in Scala using a similar technique, enabled by the feature of implicit classes [21] as shown in Figure 11.

[9] Static methods in interfaces is a feature introduced in Java 8.

```
static class Stream {
  public static Func<F, App<C, T>> OfArray<F, C, T>(T[] array)
                  where F : IStreamAlg<C>
  { return alg => alg.OfArray(array); }

  public static Func<F, App<C, T>> Filter<F, C, T>(this Func<F, App<C, T>> streamF,
                                              Func<T, bool> predicate)
                  where F : IStreamAlg<C>
  { return alg => alg.Filter(streamF(alg), predicate); }

  public static Func<F, App<E, int>> Count<F, E, C, T>(this Func<F, App<C, T>> streamF)
                  where F : IExecStreamAlg<E, C>
  { return alg => alg.Count(streamF(alg)); }

  public static App<E, int> Example<E, C, F>(int[] data, F alg)
                  where F : IExecStreamAlgebra<E, C> {
    Func<F, App<E, int>> streamF =
          Stream.OfArray(data)
                .Filter(x => x % 2 == 0)
                .Count();
    return streamF(alg);
  }
}
```

Figure 10 Example of Fluent API creation in C#.

7.2 Generalized Algebraic Data Types

We observe that the encoding of type-constructor parameterization that we employed in Section 4 is sufficient for emulating Generalized Algebraic Data Types (GADTs) in Java. GADTs are algebraic data types that permit data constructors to specify their exact return type. The standard example of a GADT is a generic expression evaluator, which can be captured via an abstract visitor that uses type-constructor polymorphism:

```
  abstract class Visitor<R<_>> {
    abstract     R <Integer> caseIntLit (IntLit expr);
    abstract     R <Boolean> caseBoolLit (BoolLit expr);
    abstract     R <Integer> casePlus (Plus expr);
    abstract <Y> R <Y>       caseIf  (If<Y> expr);
  }
```

The emulation of GADTs with type-constructor polymorphism is not new – it is, for instance, mentioned by Altherr and Cremet [1] and by Oliveira and Cook [13].

As discussed in Section 4 the safety of the emulation depends on following the convention that the only subtype of `App<S.t, ...>` is `S`.

8 Related Work

"Stream programming" is an overloaded term, encompassing streaming algorithms (a subarea of the theory of algorithms), synchronous dataflow, reactive systems, signal processing applications, spreadsheets, and embedded systems [4, 19, 24]. These are conceptually related to our work in terms of intuitions, but hardly related in terms of the techniques employed. A more appropriate context for our work is streams in the sense of the Java Stream API [15], which provides stream-like functionality for data collections. In the rest of this section, we discuss related work both from dedicated streaming systems, and from streaming APIs built on top of general-purpose programming languages.

```
object Stream {
  trait StreamAlg[C[_]] {
    def ofArray[T](array: Array[T]) : C[T]
    def filter[T](f : T => Boolean, app : C[T]) : C[T]
  }
  trait ExecStreamAlg[C[_], E[_]] extends StreamAlg[C] {
    def count[T](app : C[T]) : E[Long]
  }
  def ofArray[T, C[_], E[_], F <: ExecStreamAlg[C, E]]
             (array: Array[T]) : F => C[T] = {
    alg => alg.ofArray(array)
  }
  trait Push[T]
  trait Pull[T]
  type  Id[A] = A
  implicit class RichReader[T, C[_], E[_], F <% ExecStreamAlg[C, E]]
                            (func : F => C[T]) {
    def filter(p : T => Boolean) : F => C[T] = {
      alg => alg.filter(p, func(alg))
    }
    def count() : F => E[Long] = {
      alg => alg.count(func(alg))
    }
  }
  def example[T,  C[_], E[_]](array: Array[Int])
                              (alg : ExecStreamAlg[C, E]) : E[Long] = {
    Stream.ofArray[Int, C, E, ExecStreamAlg[C, E]](array)
          .filter((x:Int) => x%2==0)
          .count()(alg)
  }
}
```

Figure 11 Example of Fluent API creation in Scala.

Streaming DSLs and interpreters. From the implementor's point of view, our stream algebras expose a DSL for stream programming. The DSL is embedded in Java and takes advantage of the optimizing nature of the underlying JIT-based implementation. In a similar fashion, the StreamIt language, which needed a special implementation with stream-specific analyses and optimizations [23], was recently implemented atop the Java platform, as StreamJIT [3]. However, while StreamJIT required a full implementation effort, our technique is more lightweight, being available as a Java library, with an API extending that of native Java streams. StreamIt generally has a very different focus from our work: although it offers flexible, declarative combinators, its domain of applicability is multimedia streams of very large data, as opposed to general functional programming over data collections.

The DirectFlow DSL of Lin and Black also supported push and pull configurations for information-flow systems with extensible operators [9]. Compared to our design, it uses a compiler, exposes the internal "pipes" that connect different stream operators, and requires the management (instantiation and connection) of objects for these pieces of the flow graph.

The operator fusion semantics that we showed is only one example of stream-based optimizations. Other optimizations that can be unlocked by our technique are operator reordering, redundancy elimination in the pipeline, and batching [8].

Our DSL representation follows a shallow embedding with Church-style encodings [13]; as Gibbons and Wu have demonstrated, this makes it natural and simple to implement the interpreter in recursive style [5].

Our design permits not only the definition of completely new operators, but also composite

operators that reuse existing ones. This expressiveness enables modularity in our design in a manner similar to the modularity offered by the composite operators of high-level streaming languages such as SPL [7].

Collections and big data (including Java streams). Su *et al.* showed how Java streams can support different compute engines in the same pipeline [20], for the domain of distributed data sets. Unlike our design, there is no infrastructure for the change of evaluation engine without affecting the library code.

Our approach can process the pipeline, so in that respect is similar to the "application DSL" of ScalaPipe [26]. ScalaPipe operates as a program generator. It generates an AutoPipe application that produces C, OpenCL etc. The high level program is written in Scala. The target program is C.

StreamFlex offers high-throughput, low-latency streaming capabilities in Java, taking advantage of ownership types [18]. StreamFlex is an extension of Java (due to type system additions), and, furthermore, comes with an altered JVM to support real time execution. It focuses on event processing and especially on the scheduling of filters (so that priority is given to the threads that handle the stream without GC pressuring, etc.).

Compared to the above, our work is smoothly integrated in the language (as an embedded, internal DSL – effectively a plain library). We discover the DSL hidden inside the Java Streams API and show how its implementation can improve, with pluggable and modular semantics, via object algebras.

Svensson and Svenningsson demonstrated how pull-style and push-style array semantics can be combined in a single API using a defunctionalized representation and a shallow embedding for their DSL [22]. However, they propose a new API and a separate DSL layer that passes through a compiler, while we remain compatible with existing Java-like stream APIs. Furthermore, our approach enables full semantic extensibility, beyond just changing the pull vs. pull style of iteration.

9 Future Work and Conclusions

We presented an alternative design for streaming libraries, based on an object algebras architecture. Our design requires only standard features of generics and is, thus, widely applicable to modern OO languages, such as Java, Scala, and C#. We implemented a Java streaming library based on these principles and showed its significant benefits, in terms of transparent semantic extensibility, without sacrificing performance.

Given our extensible library design, there are several avenues for further work. The clearest path is towards enriching the current library implementation with shared-memory parallel evaluation semantics, cloud evaluation semantics, distributed pipeline parallelism, GPU processing, and more. Since we expose the streaming pipeline, such additions should be transparent to current evaluation semantics, and can even be performed by third-party programmers.

Acknowledgments. We thank the anonymous reviewers and artifact evaluation reviewers for their constructive comments. We are grateful to Paul Sandoz (Oracle) for help with Java 8 streams and valuable suggestions. Our work was funded by the Greek Secretariat for Research and Technology under the "MorphPL" Excellence (Aristeia) award.

References

1. Philippe Altherr and Vincent Cremet. Adding type constructor parameterization to Java. In *Workshop on Formal Techniques for Java-like Programs (FTfJP'07) at the European Conference on Object-Oriented Programming (ECOOP)*, 2007.
2. Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Clash of the lambdas. 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS), available at `http://arxiv.org/abs/1406.6631`, 2014.
3. Jeffrey Bosboom, Sumanaruban Rajadurai, Weng-Fai Wong, and Saman Amarasinghe. StreamJIT: A commensal compiler for high-performance stream programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA'14, pages 177–195, New York, NY, USA, 2014. ACM.
4. Yoonseo Choi, Yuan Lin, Nathan Chong, Scott Mahlke, and Trevor Mudge. Stream compilation for real-time embedded multicore systems. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'09, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.
5. Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP'14, pages 339–347, New York, NY, USA, 2014. ACM.
6. Daniel Gronau. HighJ – Haskell-style type classes in Java. `https://github.com/DanielGronau/highj`, 2011.
7. M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K.-L. Wu. IBM Streams Processing Language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3-4):7:1–7:11, May 2013.
8. Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys*, 46(4):46:1–46:34, March 2014.
9. Chuan-kai Lin and Andrew P. Black. DirectFlow: A domain-specific language for information-flow systems. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, volume 4609 of *ECOOP'07*, pages 299–322. Springer Berlin Heidelberg, 2007.
10. Sandro Magi. Abstracting over type constructors using dynamics in C#. `http://higherlogics.blogspot.ca/2009/10/abstracting-over-type-constructors.html`, Oct 2009.
11. Erik Meijer. The world according to LINQ. *Communications of the ACM*, 54(10):45–51, October 2011.
12. Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD'06, pages 706–706, New York, NY, USA, 2006. ACM.
13. Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses: Practical extensibility with object algebras. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, volume 7313 of *ECOOP'12*, pages 2–27. Springer Berlin Heidelberg, 2012.
14. Bruno C. d. S. Oliveira, Tijs van der Storm, Alex Loh, and William R. Cook. Feature-oriented programming with object algebras. In Giuseppe Castagna, editor, *Proceedings of the 27th European Conference on Object-Oriented Programming*, number 7920 in ECOOP'13, pages 27–51. Springer Berlin Heidelberg, January 2013.

15 Oracle. Stream (Java Platform SE 8). http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html.
16 John Rose. Hotspot-compiler-dev mailing list: Perspectives on Streams Performance. http://mail.openjdk.java.net/pipermail/hotspot-compiler-dev/2015-March/017278.html, March 2015.
17 Aleksey Shipilev, Sergey Kuksenko, Anders Astrand, Staffan Friberg, and Henrik Loef. OpenJDK: jmh. http://openjdk.java.net/projects/code-tools/jmh/.
18 Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. StreamFlex: High-throughput stream programming in Java. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, OOPSLA'07, pages 211–228, New York, NY, USA, 2007. ACM.
19 Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.
20 Xueyuan Su, Garret Swart, Brian Goetz, Brian Oliver, and Paul Sandoz. Changing engines in midstream: A java stream computational model for big data processing. *Proc. VLDB Endow.*, 7(13):1343–1354, August 2014.
21 Josh Suereth. Scala documentation: Implicit classes. http://docs.scala-lang.org/overviews/core/implicit-classes.html.
22 Bo Joel Svensson and Josef Svenningsson. Defunctionalizing push arrays. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-Performance Computing*, pages 43–52. ACM, 2014.
23 William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC'02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.
24 Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. Stream processing with a spreadsheet. In Richard Jones, editor, *Proceedings of the 28th European Conference on Object-Oriented Programming*, volume 8586 of *ECOOP'14*, pages 360–384. Springer Berlin Heidelberg, 2014.
25 Philip Wadler. The expression problem. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt, December 1998.
26 Joseph G. Wingbermuehle, Roger D. Chamberlain, and Ron K. Cytron. ScalaPipe: A streaming application generator. In *Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, FCCM'12, pages 244–, Washington, DC, USA, 2012. IEEE Computer Society.
27 Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In *Functional and Logic Programming*, pages 119–135. Springer, 2014.

Lightweight Support for Magic Wands in an Automatic Verifier

Malte Schwerhoff and Alexander J. Summers

ETH Zurich, Switzerland
`{malte.schwerhoff,alexander.summers}@inf.ethz.ch`

Abstract

Permission-based verification logics such as separation logic have led to the development of many practical verification tools over the last decade. Verifiers employ the *separating conjunction* $A * B$ to elegantly handle aliasing problems, framing, race conditions, etc.

Introduced along with the separating conjunction, the *magic wand* connective, written $A \mathrel{-\!\!*} B$, can describe hypothetical modifications of the current state, and provide guarantees about the results. Its formal semantics involves quantifying over states: as such, the connective is typically not supported in automatic verification tools. Nonetheless, the magic wand has been shown to be useful in by-hand and mechanised proofs, for example, for specifying loop invariants and partial data structures.

In this paper, we show how to integrate support for the magic wand into an automatic verifier, requiring low specification overhead from the tool user, due to a novel approach for choosing *footprints* for magic wand formulas automatically. We show how to extend this technique to interact elegantly with common specification features such as recursive predicates. Our solution is designed to be compatible with a variety of logics and underlying implementation techniques.

We have implemented our approach, and a prototype verifier is available to download, along with a collection of examples.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Magic Wand, Software Verification, Automatic Verifiers, Separation Logic, Implicit Dynamic Frames

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.614

Supplementary Material ECOOP Artifact Evaluation approved artifact available at
`http://dx.doi.org/10.4230/DARTS.1.1.10`

1 Introduction

Permission-based verification logics, most notably separation logic [24], have been widely developed in recent years, both to explore their theoretical properties and to serve as the bases for a variety of practical tools. The most well-known feature of separation logic is its *separating conjunction* connective, $*$. An assertion of the form $A * B$ intuitively expresses that the two conjuncts hold for *separate* portions of the program heap; such an assertion is true in a program state σ, if we can *split* the state into two parts, $\sigma = \sigma_1 \uplus \sigma_2$ such that A is true in σ_1 and B in σ_2. Here, $\uplus$ denotes the combination of two compatible partial program states; in particular, the two parts must describe disjoint heap locations. Support for this connective has been used to handle aliasing, framing, race conditions etc., both in by-hand proofs, and in a variety of tools.

The separating implication, or *magic wand* connective $\mathrel{-\!\!*}$ was originally introduced along with the separating conjunction, in the first papers on separation logic. The semantics of

© Malte H. Schwerhoff and Alexander J. Summers;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 614–638

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

this connective is defined as follows:

$$\sigma \models A \mathrel{-\!\!*} B \;\Leftrightarrow\; \forall \sigma' \perp \sigma \cdot\, (\sigma' \models A \;\Rightarrow\; \sigma \uplus \sigma' \models B)$$

Here, $\sigma' \perp \sigma$ expresses that the two states are compatible: neither do both require access to the same heap location, nor do they disagree on the values of any local variables. Informally, an assertion of the form $A \mathrel{-\!\!*} B$ can be understood as describing the effect of a hypothetical addition to the state σ, in the above: "if we add on any partial heap satisfying A, then B will hold in the resulting state". The ability to express guarantees about hypothetical (future) additions to the state, makes the magic wand well-suited for concisely specifying *partial* versions of data structures, e.g. for describing ongoing traversals of those structures [34, 22], or for allowing clients to reason about "recombining" a view on the whole data structure while hiding the internal definitions, which has been used for specifying protocols that enforce orderly modifications of data structures [17, 10, 15]. Yang [35] employs the magic wand for a by-hand proof of the Schorr-Waite graph marking algorithm, while Dodds et al. employ it for specifying synchronisation barriers for deterministic parallelism [9].

Despite its history and this variety of applications, the magic wand connective is generally not supported in automatic verifiers built upon separation logic (and related) theories [1, 8, 14, 21]. The quantification over states in the wand's semantics makes the connective challenging. Recent developments in *propositional* separation logics [19, 13] show its proof theory to be intricate. In the presence of variables and other logical features arising in program verification, reasoning without any user guidance is known to be undecidable [5].

We address the problem of magic wand support in the context of a general-purpose verifier, via lightweight user annotations and a novel approach for automatically choosing suitable *footprints* for magic wand assertions. We describe our solution in the context of imperative code annotated with generic user-defined predicate and function definitions (for describing program data structures). Our technique is defined in terms of core operations which most verification tools for separation logic (and similar permission-based logics) already support; it is designed to be easily implementable as an extension to existing tools. The specification language supported by our prototype implementation [29] is richer than the fragment used in this paper, and includes fractional permissions [3], quantifiers and custom domains such as mathematical sequences and sets.

Contributions

This paper shows how to support the magic wand connective in an automated verifier, including the following specific contributions:

- A design for the representation of wands in a verification state, and the provision of suitable *ghost operations* for directing their use (Section 3.2).
- An automatic strategy and algorithm for choosing suitable *footprints* for magic wand instances, without additional user direction (Section 3.4).
- A mechanism for integrating existing ghost operations (such as *folding* predicates) with our automatic footprint computation, and a soundness argument for the presented algorithms. (Section 4).
- A set of additional heuristics, which aim to infer the magic-wand-related annotations required by our approach (Section 5).
- An implementation of our techniques [29], built as an extension of an existing verification tool, along with examples demonstrating the conciseness and versatility of our approach (Section 6).

```
0  var val: Int
1  var next: Ref
2
3  predicate List(ys: Ref) {
4    acc(ys.val) * acc(ys.next) * (ys.next ≠ null ⇒ List(ys.next))
5  }
6
7  function sum_rec(ys: Ref): Int
8    requires List(ys)
9  {
10   unfolding List(ys) in
11     (ys.val + (ys.next = null ? 0 : sum_rec(ys.next))) }
12
13 method sum_it(ys: Ref) returns (sum: Int)
14   requires ys ≠ null * List(ys)
15   ensures List(ys) * sum = old(sum_rec(ys))
16 {
17   var xs := ys
18   sum := 0
19
20   while (xs ≠ null)
21     invariant ((xs ≠ null) ⇒ List(xs)) *
22       sum = (old(sum_rec(ys)) - (xs = null ? 0 : sum_rec(xs)))
23   {
24     unfold List(xs)
25     sum := sum + xs.val
26     xs := xs.next
27   }
28   // postcondition error: no permission List(ys) available
29 }
```

Figure 1 Running sum example (with insufficient loop invariant).

Our work is agnostic as to the implementation of the underlying verifier, and is designed to be easily adaptable to related verificaton logics (Section 3.1).

2 Background and Motivation

We present our work using *implicit dynamic frames* [30] as the specification logic. It provides permission-based reasoning similar to separation logic (which can be encoded [26]), and is suitable for verification both by tools based on verification-condition-generation [31, 21] and verifiers built around symbolic execution [32, 16]. We present examples in the Silver intermediate verification language [16]; our implementation extends the existing verifier Silicon for this language.

2.1 Running Example

Figure 1 shows a simple example Silver program used as our running example: a straightforward iterative implementation to calculate the sum of the nodes in a linked list. In this subsection, we give a high-level overview of the concepts involved in the specification and attempted verification of this example.

Specifications (such as the precondition marked with `requires` or the declared loop invariant) express not only the intended functional properties of the code, but also the required *permissions*; it is a general requirement that heap (field) locations may only be dereferenced if corresponding permissions are currently held. Permission to access a single field location is denoted by assertions such as `acc(ys.val)`, while permission to access an unbounded number of field locations can be expressed using predicate instances such as

```
0   var xs := ys;
1   sum := 0
2
3   define A (xs ≠ null ⇒ List(xs))
4   define B List(ys)
5
6   package A -* B
7
8   while (xs ≠ null)
9     invariant (xs ≠ null ⇒ List(xs)) * (A -* B) *
10      sum = old(sum_rec(ys)) - (xs = null ? 0 : sum_rec(xs))
11  {
12    wand w := A -* B // give magic wand instance the name w
13
14    var zs := xs // value of xs at start of iteration
15    unfold List(xs)
16    sum := sum + xs.val
17    xs := xs.next;
18
19    package A -* (folding List(zs) in (applying w in B))
20  }
21  apply A -* B
```

Figure 2 Our verified version of the body of `sum_rec`, from Figure 1.

`List(ys)`, which requires permission to all `val` and `next` fields of the linked-list beginning from `ys`. For example, the precondition of the method `sum_it` requires an instance of the `List` predicate, and its postcondition promises that such an instance will be returned to the caller, along with the guarantee that the returned value is the sum of the values stored in the list.

The verification of the while loop (line 20) relies on the provided loop invariant (line 21), which also specifies which permissions are carried along in the invariant. At the beginning of each iteration of the loop body, an `unfold` annotation (line 24) directs the verifier to unroll the (recursive) definition of the `List` predicate instance. According to the definition of the predicate (line 3), this makes available the permissions to access the fields of `xs`, which is necessary for the verifier to allow the subsequent assignments (lines 25 and 26).

After the loop (line 28) the verifier will have a copy of the state described by the loop invariant, as well as the assumption that the loop condition is false. Unfortunately, in this case, the provided loop invariant provides no permissions; essentially, the `List` predicate instance has been totally unfolded during traversal of the list, and the left-over permissions were not retained in the loop invariant. As a consequence, the postcondition (line 15) will fail to verify, since the required predicate instance cannot be found.

2.2 Overview of Magic Wand Support

Figure 2 shows the body of the `sum_it` method, specified using our magic wand support. The loop invariant has been strengthened (line 9) to include an additional *magic wand instance*[1] `(xs ≠ null ⇒ List(xs)) -* List(ys)`.

Informally, this magic wand instance represents the following promise: "if you give up permission to the remainder of the list (starting from `xs`), I will give you back permission to the entire list structure (starting from `ys`)". This assertion plays the role of representing the

[1] We have used syntactic abbreviations (lines 3 and 4) to make the code more readable, and to save repetition of these assertions; they are also supported in our tool.

permissions to the partial list inspected so far by the loop; we say these permissions make up the *footprint* of the magic wand.

The footprint of a magic wand must include enough permissions to make this informal promise justified. We can direct the verifier to create a new magic wand instance (and choose a suitable such footprint) using a `package` statement, such as that on line 6, which creates the wand instance necessary for showing that the loop invariant holds on entry. An empty footprint suffices on line 6, since `xs` and `ys` are equal at this point (line 0).

During verification of the loop body, we need to maintain the magic wand instance in the loop invariant; this is achieved by the `package` statement on line 19, which produces a new suitable magic wand instance to represent the current left-over permissions to the already-inspected portion of the list. Apart from the assertions A and B, the extra annotations on this line explain to the verifier *how*, given the left-hand-side assertion, the right-hand-side assertion can be obtained[2]. Given these annotations, the calculation of the extra permissions which must be associated with this new wand instance (i.e. its footprint) is performed automatically; our techniques for achieving this are an important contribution of this paper.

A wand instance can be combined with its left-hand-side (LHS) assertion, and the combination exchanged for the right-hand-side (RHS) assertion; this is called *applying* the magic wand instance. For example, after the loop body in our example, the magic wand instance from the loop invariant is *applied* (on line 21); its LHS must be given up, and its RHS `List(ys)` is added to the state, providing the method's postcondition.

In the rest of the paper, we use this simple example to help explain the details of our general magic wand support: the representation of magic wands, related annotations, and our automatic footprint computation algorithm (Section 3); the integration of other ghost operations such as `folding` on line 19 (Section 4); and a set of heuristics used to infer magic-wand-related annotations (Section 5). In the remainder of this section, we provide more-detailed background and foundational definitions.

2.3 Assertion Language

The assertion language used in this paper consists of the following constructs:

$$A ::= e \mid \texttt{acc}(e.f) \mid P(e) \mid A * A \mid e \Rightarrow A \mid A \mathrel{-\!\!*} A$$

This is a core fragment of the assertions employed in the Silver language, extended (in the last case) with *magic wand assertions*; A denotes assertions, e denotes side-effect-free expressions. Permissions are managed in the logic via *accessibility predicates* $\texttt{acc}(e.f)$, which denote the *exclusive* permission to access a heap location $e.f$. For example, see line 4 of Figure 1. The conjunction $*$ behaves as the separating conjunction in separation logics; in particular, an assertion $\texttt{acc}(x.f) * \texttt{acc}(y.f)$ requires the *disjoint union* of the permissions required by the two conjuncts; this implicitly requires that $x \neq y$, otherwise the same (exclusive) permission would be required twice (this is analogous to the meaning of the assertion $x.f \mapsto _ * y.f \mapsto _$ in separation logic). The grammar above imposes the standard restriction [1] that accessibility predicates (as well as predicate and magic wand instances) may not occur on the left of conditional assertions: the value of the condition e is therefore independent of the current permissions held.

[2] Variable `zs` records the node that `xs` pointed to at the beginning of the current loop iteration, while `w` gives a name to the magic wand instance belonging to the loop invariant at the start of the iteration. Both are not strictly necessary, but make the annotations on line 19 succinct.

In contrast to separation logic, implicit dynamic frames allows *heap-dependent expressions* such as $x.f_1.f_2 > 0$ to be used in assertions. In particular, heap-dependent *functions* can be defined and used in expressions, such as the `sum_rec` function in Figure 1 (line 7). Heap-dependent expressions are only guaranteed a meaningful semantics when they are *framed* by the permissions held in the state in which they are evaluated, meaning that for any heap location dereferenced by the expression, a corresponding permission must currently be held. Accessing heap locations in program statements is similarly restricted; e.g. a field read such as `xs.val` on line 25 of Figure 1 is allowed only in states in which a permission to the location `xs.val` is held. An assertion is said to be *self-framing* if it requires at least permissions to those locations on which the expressions it mentions depend. For example, the assertion `xs.val > 0` is not self-framing, whereas `acc(xs.val) * xs.val > 0` is self-framing. Invocations of functions such as `sum_rec` must analogously occur in states in which their preconditions (e.g. line 8) hold. Only self-framing assertions can be used in specifications. As a technical simplification of this check, we assume that the permission to access a heap location comes syntactically *before* any expressions depending on the value at that location (this is analogous to the restriction in some separation-logic-based tools that logical variables must be bound to heap locations before their use). In [26], Parkinson and Summers have shown that separation logic assertions can be encoded into implicit dynamic frames, and that the resulting assertions are self-framing by construction.

To simplify the presentation of our algorithms, we restrict ourselves in this paper to unary predicates $P(e)$ and functions $g(e)$ (the details of which will be discussed in Section 2.5), but this arity restriction is not relevant for the techniques presented in this paper. Our implementation [29] supports unrestricted predicate and function definitions, as well as fractional permissions [3].

2.4 Verification via Exhale and Inhale Operations

From a verification perspective, proof obligations can be expressed in Silver via *exhaling* and *inhaling* assertions [21], which are permission-aware analogues of traditional *assume*/*assert* statements used to express verification conditions. Analogous operations are used internally in other verification tools (e.g. for separation logic); in tools based on symbolic execution, these operations are typically called *consume* and *produce*.

An operation `exhale` A (where A is an assertion) can be understood to *assert* all of the logical properties described by A, and to *give away* all of the permissions described by the assertion. Once permissions have been given away, the verifier may no longer retain (or *frame*) facts about the values of these heap locations, even if permission is regained later. For example, before the while loop in our running example, the loop invariant is exhaled; the giving up of permissions reflects the fact that the loop may modify the locations to which the loop invariant requires access. The loop invariant must also be exhaled at the end of the loop body, reflecting the usual requirement that the invariant is preserved.

In terms of the state maintained by a verifier, an `exhale` operation can be understood as requiring the current verification state σ to be *split* into a part σ_1 satisfying the assertion A, and a remainder state σ_2, which is the result of the operation. From a soundness perspective, it is fine for a verifier to overapproximate σ_1, effectively giving more permissions away than is necessary; the precision of these operations depends on the completeness of the underlying tool. If such a split *cannot* be found (the assertion A could not be shown to hold in the original state), then this operation causes a verification error, similar to an assertion failure in first-order tools.

`inhale` A is the dual operation: it *assumes* the logical properties described by A, and

adds the permissions to the current state. As an operation on states, this can be regarded as *combining* an arbitrary state satisfying A with the current state. The verification of high-level programming features can typically be modelled using combinations of these operations: for example, a method call can be modelled by *exhaling* the method's precondition, and *inhaling* its postcondition. In terms of Figure 2, the loop invariant (line 9) is inhaled at the beginning of checking the loop body (line 11) as well as after the loop, for verifying the subsequent code (line 21).

2.5 Recursive Definitions and Ghost Operations

In Silver, unbounded data structures can be specified via *recursive predicates* [25], such as `List`. An instance of this predicate (written e.g. `List(xs)`) represents permissions to all directly and transitively (via `next`) reachable fields[3]. A predicate definition can have any number of parameters, and its body may be any self-framing assertion; in particular, it may include instances of the same or other predicates, and express conditions over arbitrary combinations of the parameter and heap values accessed.

Complete reasoning in the presence of such predicates is undecidable; consequently, many automatic verifiers do not treat a predicate instance as simply a direct short-hand for its body (the *equi-recursive* interpretation [33]). Instead, tools typically differentiate between holding an instance of a predicate and holding the assertion defined by its body, while allowing the two to be exchanged via `unfold` and `fold` operations (the *iso-recursive* interpretation). An `unfold` operation directs that a currently-held predicate instance should be exchanged for its body, while a `fold` operation exchanges the body for a predicate instance. Until an `unfold` is specified, predicate instances are treated as *opaque*, in the sense that the permissions and logical facts entailed by their definitions are not directly available to the verifier. We call such operations (which rewrite the verification state to guide the tool, but do not involve changes to the program state) *ghost operations*.

In some tools, ghost operations must be explicitly specified within the program code, while other tools may attempt to infer these via heuristics/static analyses. For the purposes of this paper, we will include ghost operations as explicit statements in the program text[4].

As alluded to above, Silver also supports recursive side-effect-free *functions* in specifications. In our example, the function `sum_rec` returns the sum of the integer values stored in the linked-list. A Silver function's body is an *expression*; the function's precondition must require enough permissions to guarantee that the function's body is framed (in the case of `sum_rec`, it requires an instance of the `List` predicate). The body of the `sum_rec` function computes the sum in the natural recursive manner; the only non-standard feature is the `unfolding`. This construct does not affect the value returned by the function: its role is to tell the verifier to *temporarily* apply an `unfold` ghost operation *before* evaluating the nested expression (after the "`in`"), explaining how to find the necessary permissions.

2.6 Revisiting the Running Example

Armed with the above background, we can explain the usage of magic wands in Figure 2 more clearly. In particular, the magic wand in the loop invariant retains the appropriate

---

[3] Technically, recursive predicate definitions should be understood via their least fixpoint interpretations. We do not concern ourselves with well-definedness details regarding recursive definitions, which are not the main focus of our paper.

[4] However, in Section 5 we will show how we can indeed infer these in many cases.

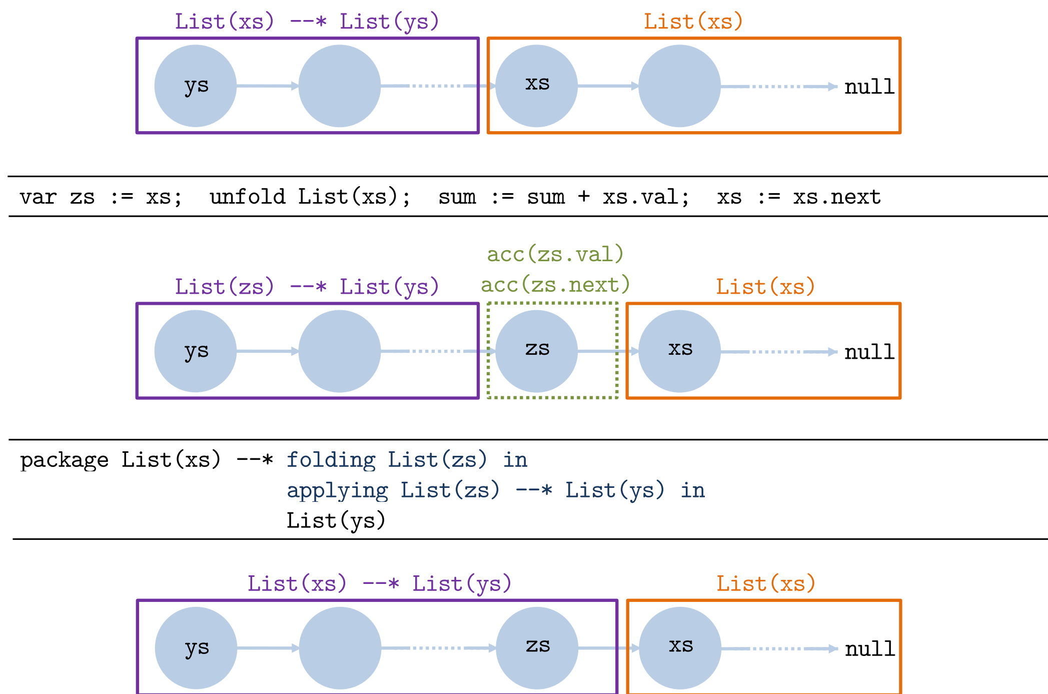


Figure 3 Illustration of the organisation of permissions in the loop invariant from Figure 2, via magic wand and predicate instances. The magic wand instance covers permissions to the prefix of the list (starting at `ys`) that has already been traversed by the loop. For simplicity, the cases of `xs`/`xs.next` being `null` have been ignored.

permissions to the already-inspected nodes in the original list, such that rather than these being lost after the loop terminates, they can be recovered by simply applying the wand (line 21). This enables the verification of the postcondition of `sum_it` (Figure 1, line 15), which expresses that a `List` predicate will be returned to the caller, along with the knowledge that this iterative code computes the same value as the function `sum_rec`[5].

Figure 3 conceptually illustrates the permissions that the magic wand instance in our loop invariant represents, by stepping through the important stages of verifying the loop body (for simplicity, the cases of `xs`/`xs.next` being `null` have been ignored). At the beginning of the loop body, the magic wand's footprint includes the permissions (to fields `val` and `next`) from the head of the linked list `ys` all the way down to – but excluding – the current node `xs`. The remaining permissions, i.e. those to the current node and the tail of the list, are contained in the predicate instance `List(xs)`. The latter is then unfolded, providing permissions to the fields of `xs`, i.e. to `acc(xs.val)` and `acc(xs.next)`, and `xs` is then advanced such that it points to the next node (i.e. to `zs.next`). In order to reestablish the loop invariant, in particular, to reestablish that the wand instance includes the permissions to the already visited prefix of the list, we therefore need to add the permissions to `zs.val` and `zs.next` to the wand instance. This is (conceptually) achieved by the final `package`-statement: the ghost operations on the RHS of the wand force the wand's footprint to include the footprint of the wand held so far, plus the permissions to `zs.val` and `zs.next` (necessary for the `folding` ghost operation specified). Ghost operations will be explained in Section 4.

[5] The `old` construct specifies that the nested expression should be evaluated in the *heap* of the method's prestate; the evaluation of program variables is not affected by `old`.

As we show in this paper, our support for the magic wand connective allows a natural specification of these “left-over” parts of data structures, in a way which requires few annotations, and applies equally well to other data structures and predicates. This is an important use-case for our work, but (as discussed in the introduction) the magic wand has various other known applications (e.g. [17, 10, 15, 35, 9]); the possibility of practical tool support via the contributions of this paper will likely also lead to further applications being explored.

3 Magic Wand Support with Automatic Footprints

We present our solution for supporting the magic wand without relying on any particular implementation strategy for the underlying verification tool. For example, we are agnostic as to whether the verifier is based on symbolic execution, verification condition generation, or some other technique, so long as the modelled program state admits a number of basic operations presented in the next subsection. Moreover, although we present our approach in the context of implicit dynamic frames, it is straightforward to adapt it to a separation-logic-based tool or to other permission-based verification logics.

3.1 Basic Operations

We use σ to range over program states as modelled in the verifier. We do not prescribe a particular representation for these states; in a tool based on symbolic execution, they could be sets of heap chunks along with path conditions, while in a tool based on verification condition generation, they could consist of maps representing the heap and permissions held. States must be able to record assumptions, permissions and magic wand instances (see the next subsection). Figure 4 defines the interface we expect to be implemented by the states. We represent these interface operations as functions on (and producing) immutable states. In practice, the operations could be implemented by generating a corresponding program in an intermediate language, or by directly updating internal (potentially mutable) state in a verification tool.

The state operations `hasAcc`, `addAcc` and `removeAcc` are used respectively to check that a state holds permissions to a field, to add and to remove such permission from a state. Analogous operations are included for predicate and magic wand instances. With respect to our running example (Figure 2), `addPred` will be, for example, used to add an instance of `List(xs)` to the state when inhaling the loop invariant at the beginning of the loop body, and `removePred` will be used at the end of the loop body when exhaling the loop invariant. The `unfold` operation on line 15 would typically use (amongst other operations) `removePred` to remove the predicate instance `List(xs)` that is to be unfolded, and `addAcc` to add permissions to `xs.val` to the state. In line 16, `hasAcc` would be used to assert that reading the field `xs.val` is permitted, i.e. that the state holds permissions to the field.

Depending on the implementation approach taken for a particular verifier, the implementation of these operations will vary. In a verifier which translates to an intermediate language such as Boogie [20], operations such as `addAcc` and `removeAcc` would typically be implemented by generating modifications of the Boogie program state (used to model e.g. permissions held), while in a symbolic execution tool these operations would typically involve mutation of a collection maintained by the verification tool to represent the make up of the current state.

The idea behind `equate` is to be able to communicate information (i.e. logical constraints) from one state to another. In particular, we use this operation to model adding the information

σ.`eval`(e)	$\approx$ evaluates expression e in state σ, yielding a value v
σ.`assume`(e)	$\approx$ assume that e holds in σ
σ.`assert`(e)	$\approx$ assert that e holds in σ
σ.`hasAcc`(v, f)	$\approx$ true iff σ contains access to $v.f$
σ.`addAcc`(v, f)	$\approx$ add access to $v.f$ to σ
σ.`removeAcc`(v, f)	$\approx$ remove access to $v.f$ from σ
σ.`hasPred`(P, v)	$\approx$ true iff σ contains $P(v)$
σ.`addPred`(P, v)	$\approx$ add $P(v)$ to σ
σ.`removePred`(P, v)	$\approx$ remove $P(v)$ from σ
σ.`hasWand`$(A \mathbin{-\!\!*} B)$	$\approx$ true iff σ contains $A \mathbin{-\!\!*} B$
σ.`addWand`$(A \mathbin{-\!\!*} B)$	$\approx$ add $A \mathbin{-\!\!*} B$ to σ
σ.`removeWand`$(A \mathbin{-\!\!*} B)$	$\approx$ remove $A \mathbin{-\!\!*} B$ from σ
σ.`onlyvars`$()$	$\approx$ returns a state σ' that is empty, except for local variables
	$\approx$ declared in σ, and all assumptions σ has about them
σ_1.`equate`(σ_2, v, f)	$\approx$ update σ_1 s.t. it contains all assumptions from σ_2 about $v.f$
`if` $(\ldots)$ $\ldots$ `else` $\ldots$	$\approx$ conditional operation

Figure 4 Basic state operations. e denotes an expression, σ a state, and v denotes a value (of appropriate type), i.e. the result of evaluating an e in a σ. All operations except `eval`, `hasAcc`/`hasPred`/`hasWand`, `onlyvars` and `if` return an updated state.

that the value of an expression in σ_1 is the same as in σ_2. We expect σ_1.`equate`(σ_2, v, f) to produce a modified version of σ_1, in which information known about the value of $v.f$ in σ_2 has been copied/made available. In practice, this operation often has a simple implementation; it could amount simply to equating the symbolic values of $v.f$ in the two states, or (in a tool based on verification condition generation) simply adding the assumption that the value of the expression is the same in the two states. In implementations in which logical constraints (path conditions) are not stored globally, the operation might require selectively copying such constraints.

The `if` conditional can be implemented differently in different tools: those which translate to another language for verification (such as Boogie) may represent this as an actual conditional, whereas a symbolic-execution-based verifier would typically *branch* (that is, split the proof of the program) at this point. We overload `if` such that it can be used with (boolean) state values v, e.g. `if` σ.`eval`(e), and with (boolean) return values of operations on state, e.g. `if` σ.`hasAcc`(v, f).

We can now define the `inhale` and `exhale` operations as functions of states (Figure 5), in terms of the basic operations above. We use the $\leadsto$ symbol to denote the desugaring/compilation of an operation into simpler ones. For example, σ.`inhale`$(a_1 * a_2) \leadsto \sigma$.`inhale`$(a_1)$.`inhale`$(a_2)$ represents that inhaling a conjunction is defined as inhaling the second conjunct in the state resulting from inhaling the first conjunct.

$$
\begin{array}{lcl}
\sigma.\texttt{exhale}(a) & \rightsquigarrow & \sigma.\texttt{exhale}(\sigma, a) \\
\sigma.\texttt{exhale}(\widetilde{\sigma}, e) & \rightsquigarrow & \widetilde{\sigma}.\texttt{assert}(e) \\
\sigma.\texttt{exhale}(\widetilde{\sigma}, a_1 * a_2) & \rightsquigarrow & \sigma.\texttt{exhale}(\widetilde{\sigma}, a_1).\texttt{exhale}(\widetilde{\sigma}, a_2) \\
\sigma.\texttt{exhale}(\widetilde{\sigma}, e \Rightarrow a) & \rightsquigarrow & \texttt{if } \widetilde{\sigma}.\texttt{eval}(e)\ \sigma.\texttt{exhale}(\widetilde{\sigma}, a) \texttt{ else } \sigma \\
\sigma.\texttt{exhale}(\widetilde{\sigma}, \texttt{acc}(e.f)) & \rightsquigarrow & \\
\quad v \texttt{ := } \widetilde{\sigma}.\texttt{eval}(e) & & \\
\quad \texttt{if } \sigma.\texttt{hasAcc}(v, f)\ \sigma.\texttt{removeAcc}(v, f) \texttt{ else fail} & &
\end{array}
$$

$$
\begin{array}{lcl}
\sigma.\texttt{inhale}(e) & \rightsquigarrow & \sigma.\texttt{assume}(e) \\
\sigma.\texttt{inhale}(a_1 * a_2) & \rightsquigarrow & \sigma.\texttt{inhale}(a_1).\texttt{inhale}(a_2) \\
\sigma.\texttt{inhale}(e \Rightarrow a) & \rightsquigarrow & \texttt{if } \sigma.\texttt{eval}(e)\ \sigma.\texttt{inhale}(a) \texttt{ else } \sigma \\
\sigma.\texttt{inhale}(\texttt{acc}(e.f)) & \rightsquigarrow & v \texttt{ := } \sigma.\texttt{eval}(e);\ \sigma.\texttt{addAcc}(v, f)
\end{array}
$$

Figure 5 The interesting cases for the definitions of exhaling and inhaling assertions (see also [21]). The second state parameter $\widetilde{\sigma}$ for `exhale` is used to carry a copy of the original state, used when checking boolean expressions (to avoid any loss of information due to removed permissions). The cases for inhaling/exhaling predicate and wand instances are analogous to the case of inhaling/exhaling `acc`$(e.f)$. `fail` is a short-hand for σ.`assert(false)`.

3.2 Representing, Applying and Packaging Wands

Our approach to supporting magic wands can be related to the handling of recursive definitions via ghost operations (cf. Section 2.5) as follows: Just as for predicates, we wish to be able to derive new magic wand assertions, and to apply their meanings while verifying code, but tackling this problem automatically without any direction from the user is known to be undecidable, even for much more restricted assertion logics than those we wish to support [5].

Analogous to predicate instances, we treat instances of magic wands as *opaque*; when one is available in a verification state, the verifier need not attempt to deduce anything that follows from the wand's meaning, without direction to do so. The choice to use such a magic wand instance must be directed by a ghost statement `apply` $A \mathrel{-\!\!*} B$ (see, for example, line 21 of Figure 2). Recall the formal semantics of a magic wand assertion from Section 1:

$$\sigma_{foot} \vDash A \mathrel{-\!\!*} B \;\Leftrightarrow\; \forall \sigma_A \perp \sigma_{foot} \cdot\ (\sigma_A \vDash A \;\Rightarrow \sigma_{foot} \uplus \sigma_A \vDash B)$$

This semantics intuitively says that $A \mathrel{-\!\!*} B$ is true in a state σ_{foot} if it is guaranteed that the state created by combining this state with some additional state σ_A satisfying A, satisfies B. One can see this as a definition in terms of what can be *deduced* from a magic wand, according to the following *Modus-Ponens*-like inference rule from separation logic: $A * (A \mathrel{-\!\!*} B) \vDash B$, and we analogously define the operation of applying a wand instance in a state σ as follows:

$$\sigma.\texttt{apply}(A \mathrel{-\!\!*} B) \rightsquigarrow \sigma.\texttt{exhale}(A \mathrel{-\!\!*} B).\texttt{exhale}(A).\texttt{inhale}(B)$$

Just as for predicate instances, the opaque treatment of magic wand instances requires for soundness that the state σ_{foot} in the semantics above must notionally *belong* to the magic wand instance, in the sense that the program is not allowed to modify that part of the state up until the wand instance is applied. We call such a state the *footprint* of the magic wand instance. Whenever a new magic wand instance is to be added to the state, we need to compute some suitable part σ_{foot} of the current state σ, that will suffice to guarantee the wand's semantics, and then remove σ_{foot} from the current state, and add the new magic wand instance. We call this operation (of choosing a suitable footprint for a magic wand

instance, and exchanging the footprint for the wand instance) *packaging* the magic wand instance, and use a ghost command `package` $A \mathrel{-\!\!*} B$ to indicate that this operation should be performed.

Unlike the folding of a predicate instance, a suitable choice of footprint for a magic wand instance is not directly determined by its definition. As discussed, such a footprint must be *some* state guaranteeing the semantics of the wand, but this doesn't indicate *how* this state should be chosen. The key contribution which keeps our approach lightweight is that we have defined a useful strategy (and corresponding algorithm) for automatically choosing footprint states.

3.3 Strategy for Choosing Footprints

Automatically choosing a suitable footprint for a magic wand instance is challenging. According to the approach outlined in the previous subsection, a package operation `package` $A \mathrel{-\!\!*} B$ must attempt to choose a footprint σ_{foot}, which can be any portion of the current state so long as it satisfies the wand's semantics. In checking this criterion, it would be unsound to use any facts from the current state which are *not* framed by permissions that we choose to put into the footprint, since these might no longer be true by the time the wand instance is applied[6]. For example, when proving `acc(x.f) -* acc(x.f) * x.f = 3` in a state where `x.f = 3`, we may only use this fact if we store permission to `x.f` in the footprint; otherwise, the value of `x.f` could have been changed by the time the wand instance is applied.

In deciding on a strategy for choosing footprints, we could soundly restrict the choice of a wand instance's footprint state to be *any* portion of the current state, so long as we then check that this choice indeed guarantees the wand's semantics. Certain strategies for choosing a footprint are, however, more useful than others. For example, we *could* always choose the empty state as a footprint, and therefore not use up any permissions at a `package` statement; the check of the wand's semantics (with $\sigma_{foot} = \varnothing$) would typically then fail: effectively we would only support wands where A logically entails B, which, for example, would not be sufficient to specify our running example. Alternatively, we could always choose the *entire current state* to be the new wand instance's footprint. This would allow many wands to be proven, but would mean that the remaining program would be almost certain not to verify, since all permissions from the current state would have been lost to the wand instance. Although either of these approaches would be sound, they would not be useful in practice.

Intuitively, it makes sense to choose a footprint which is as small as possible, while still guaranteeing enough information for the wand's semantics. However, the notion of "as small as possible" is not straightforward to define precisely. For example, we can satisfy the semantics of a wand $A \mathrel{-\!\!*} B$ by choosing a footprint state which includes enough permissions such that a state satisfying A can never again be obtained; this would yield a true wand instance (the inability to find a suitable σ_A state makes the semantics of the wand vacuously true), but one which could never be applied, which is not useful as a verification construct.

Recall the previous example, in which `acc(x.f) -* acc(x.f) * x.f = 3` is to be packaged in a state in which `x.f = 3`. As discussed, this fact may only be soundly used when proving the RHS of the wand if permission to `x.f` goes into the wand's footprint. Although this extra logical fact is useful in proving the right-hand-side, such a decision would again yield a wand instance which cannot be applied, since the LHS of this wand requires this

[6] We handle only magic wand assertions $A \mathrel{-\!\!*} B$ in which the assertions A and B are self-framing. Thus, we disallow awkward assertions such as $true \mathrel{-\!\!*} x.f = 3$. This is not a strong restriction in practice; indeed, in standard separation logics, all assertions are self-framing [26].

$$
\begin{array}{l}
\texttt{transfer}(\overline{\sigma_i}\cdot\sigma, \sigma_{used}, \texttt{acc}(e.f)) \;\rightsquigarrow \\
\quad v \texttt{ := } \sigma_{used}.\texttt{eval}(e) \\
\quad \texttt{if } \sigma.\texttt{hasAcc}(v, f)\ \{ \\
\quad\quad \sigma'_{used} \texttt{ := } \sigma_{used}.\texttt{addAcc}(v, f) \\
\quad\quad \sigma''_{used} \texttt{ := } \sigma'_{used}.\texttt{equate}(\sigma, v, f) \\
\quad\quad \sigma' \texttt{ := } \sigma.\texttt{removeAcc}(v, f) \\
\quad\quad \texttt{return } (\overline{\sigma_i}\cdot\sigma', \sigma''_{used}) \\
\quad \}\ \texttt{else}\ \{ \\
\quad\quad (\overline{\sigma'_i}, \sigma'_{used}) \texttt{ := } \texttt{transfer}(\overline{\sigma_i}, \sigma_{used}, \texttt{acc}(e.f)) \\
\quad\quad \texttt{return } (\overline{\sigma'_i}\cdot\sigma, \sigma'_{used}) \\
\quad \} \\
\\
\texttt{transfer}(\epsilon, \sigma_{used}, \texttt{acc}(e.f)) \;\rightsquigarrow\; \texttt{fail}
\end{array}
$$

Figure 6 $\overline{\sigma_i}$ denotes a (potentially empty) stack of states, and $\overline{\sigma_i}\cdot\sigma$ denotes a stack created by pushing a single state σ onto a stack of states $\overline{\sigma_i}$. Function `transfer` descends a stack of states and tries to find permissions to location $e.f$. If successful, the permissions are transferred into σ_{used}. The cases for $P(e)$ and $A \mathrel{-\!\!*} B$ are defined analogously.

permission to be provided when applying the wand instance. Essentially, any permissions which we choose to take from the current state when they are already provided by the LHS of the wand, are *leaked* at the point of packaging an instance of that wand, which is not typically useful for verifying the rest of the program.

Motivated by these observations, our strategy for choosing wand footprints is: *include all permissions required by the wand's RHS, which we cannot prove to be provided by the wand's LHS*. We observe that restricting the choice of footprint to *only* these permissions is not really a restriction in practice. If the tool user *intends* to include extra permissions from the current state to in a wand's footprint, they can achieve it by writing a RHS which requires more permission than the LHS provides. In the next subsection, we explain how this high-level strategy can be realised in practice.

3.4 Footprint Computation Algorithm

The idea of our strategy is simple, but writing an algorithm that implements it is still challenging: there is a technical circularity to the problem. The footprint for a wand is determined in terms of the permissions required by its RHS. Exactly which permissions are required by the RHS can (due to implications/conditionals) depend on properties of heap values. Properties known about heap values in the *current* state may be soundly used if and only if permissions to those heap locations are included in the wand's footprint (which we are trying to compute).

To break this circularity, we devised an algorithm called `exhale_ext` to *simultaneously* evaluate the wand's RHS to determine the permissions it requires, and construct a new state σ_{used}, containing these required permissions. These permissions are taken from the current state if they cannot be proved to be provided by the wand's LHS; thus, we implicitly carve out a suitable footprint for the wand from the current state.

Our algorithm is shown in Figure 7, and works as follows: Let σ be the current state in which a `package` $A \mathrel{-\!\!*} B$ operation takes place. We first construct a hypothetical extra state σ_A representing the information provided by the wand's LHS (by inhaling the LHS into

```
σ.package(A -* B) ⇝
    σ_emp := σ.onlyvars()
    σ_A := σ_emp.inhale(A)
    (σ' · σ'_A, σ'_used) := exhale_ext(σ · σ_A, σ_emp, B)
    return σ'.addWand(A -* B)

exhale_ext(σ̄_i, σ_used, acc(e.f)) ⇝
    return transfer(σ̄_i, σ_used, acc(e.f))

exhale_ext(σ̄_i, σ_used, A_1 * A_2) ⇝
    (σ̄'_i, σ'_used) := exhale_ext(σ̄_i, σ_used, A_1)
    (σ̄''_i, σ''_used) := exhale_ext(σ̄'_i, σ'_used, A_2)
    return (σ̄''_i, σ''_used)

exhale_ext(σ̄_i, σ_used, e) ⇝
    σ_used.assert(e)
    return (σ̄_i, σ_used)
```

Figure 7 Packaging a wand instance. σ_{emp} is empty except for local variables and assumptions about those from σ. Permissions transferred into σ_{used} contribute to the footprint. `exhale_ext` of $P(e)$ and $A \mathrel{-\!\!*} B$ is defined analogous to the case of `acc`$(e.f)$. Other cases of `exhale_ext` are analogous to `exhale`, but expressions are evaluated in σ_{used}, as in the e case, above.

an empty state σ_{emp}). We then use `exhale_ext` to attempt to compute a state σ'_{used} that satisfies the wand's RHS, i.e. such that $\sigma'_{used} \vDash B$. State σ'_{used} is constructed by successively transferring permissions from the *stack* of states $\sigma \cdot \sigma_A$ into σ_{emp}. As motivated in Section 3.4, the algorithm tries to minimise the computed footprint by taking permissions preferentially from σ_A and only from σ when needed[7].

Our `exhale_ext` algorithm computes a footprint only implicitly; the state σ_{used} is not the footprint itself, but (if the algorithm terminates successfully) corresponds to (a part of) σ_A combined with a part of the input state σ which has been removed from the resulting state σ'; the removed part corresponds to the chosen footprint. At at any point *during* the execution of `exhale_ext`, the current σ_{used} satisfies the prefix of the wand's RHS that has been processed so far. Recall that both sides of a wand have to be self-framing; in particular, on a wand's RHS the permission to access a heap location occurs before any expressions that depend on the location's value. This is why expressions can be evaluated in σ_{used}; any necessary permissions (and assumptions about the location's value) will already have been transferred into this state by our algorithm.

The separation of the two initial states σ and σ_A in our algorithm is essential for a correct footprint computation. A naïve algorithm which simply *combined* the hypothetical extra state with the current state before trying to exhale the wand's RHS would be unsound: this combination might be inconsistent (due to holding too many permissions, or to conflicting value facts), which would trivialise the check of the wand's RHS assertion.

[7] In the context of Figure 7 the stack of states making up the first argument to `exhale_ext` will always consist of *two* states, corresponding to the input $\sigma \cdot \sigma_A$, but our algorithm is defined to take a stack of states of any length. This generalisation will be shown to be necessary in Section 4 for supporting nested magic wand assertions.

$\sigma.\texttt{package}(A \mathbin{-\!\!*} G) \rightsquigarrow$
$\quad \sigma_{emp} := \sigma.\texttt{onlyvars}()$
$\quad \sigma_A := \sigma_{emp}.\texttt{inhale}(A)$
$\quad (\sigma' \cdot \sigma'_A, \sigma'_{used}) := \texttt{exec}(\sigma \cdot \sigma_A, \sigma_{emp}, G)$
$\quad \texttt{return } \sigma'.\texttt{addWand}(A \mathbin{-\!\!*} \texttt{nested}(G))$

$\texttt{exec}(\overline{\sigma_i}, \sigma_{ops}, \texttt{folding}\, P(e)\, \texttt{in}\, G) \rightsquigarrow$
$\quad \sigma_{emp} := \sigma_{ops}.\texttt{onlyvars}()$
$\quad v := \sigma_{ops}.\texttt{eval}(e)$
$\quad (\overline{\sigma'_i} \cdot \sigma'_{ops}, \sigma'_{used}) := \texttt{exhale_ext}(\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{emp}, \texttt{Body}(P)[\texttt{param} \mapsto v])$
$\quad \sigma''_{used} := \sigma'_{used}.\texttt{fold}(P, v)$
$\quad \texttt{return exec}(\overline{\sigma'_i}, \sigma'_{ops} \uplus \sigma''_{used}, G)$

$\texttt{exec}(\overline{\sigma_i}, \sigma_{ops}, \texttt{unfolding}\, P(e)\, \texttt{in}\, G) \rightsquigarrow$
$\quad \sigma_{emp} := \sigma_{ops}.\texttt{onlyvars}()$
$\quad v := \sigma_{ops}.\texttt{eval}(e)$
$\quad (\overline{\sigma'_i} \cdot \sigma'_{ops}, \sigma'_{used}) := \texttt{exhale_ext}(\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{emp}, P(e))$
$\quad \sigma''_{used} := \sigma'_{used}.\texttt{unfold}(P, v)$
$\quad \texttt{return exec}(\overline{\sigma'_i}, \sigma'_{ops} \uplus \sigma''_{used}, G)$

$\texttt{exec}(\overline{\sigma_i}, \sigma_{ops}, \texttt{applying}\, A \mathbin{-\!\!*} B\, \texttt{in}\, G) \rightsquigarrow$
$\quad \sigma_{emp} := \sigma_{ops}.\texttt{onlyvars}()$
$\quad (\overline{\sigma'_i} \cdot \sigma'_{ops}, \sigma'_{used}) := \texttt{exhale_ext}(\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{emp}, A * (A \mathbin{-\!\!*} B))$
$\quad \sigma''_{used} := \sigma'_{used}.\texttt{apply}(A \mathbin{-\!\!*} B)$
$\quad \texttt{return exec}(\overline{\sigma'_i}, \sigma'_{ops} \uplus \sigma''_{used}, G)$

$\texttt{exec}(\overline{\sigma_i}, \sigma_{ops}, \texttt{packaging}\, A \mathbin{-\!\!*} G_1\, \texttt{in}\, G_2) \rightsquigarrow$
$\quad \sigma_{emp} := \sigma_{ops}.\texttt{onlyvars}()$
$\quad \sigma_A := \sigma_{emp}.\texttt{inhale}(A)$
$\quad (\overline{\sigma'_i} \cdot \sigma'_{ops} \cdot \sigma'_A, \sigma'_{used}) := \texttt{exec}(\overline{\sigma_i} \cdot \sigma_{ops} \cdot \sigma_A, \sigma_{emp}, G_1)$
$\quad \sigma''_{ops} := \sigma'_{ops}.\texttt{addWand}(A \mathbin{-\!\!*} \texttt{nested}(G_1))$
$\quad \texttt{return exec}(\overline{\sigma'_i}, \sigma''_{ops}, G_2)$

$\texttt{exec}(\overline{\sigma_i}, \sigma_{ops}, A) \rightsquigarrow$
$\quad \sigma_{emp} := \sigma_{ops}.\texttt{onlyvars}()$
$\quad \texttt{return exhale_ext}(\overline{\sigma_i} \cdot \sigma_{ops}, \sigma_{emp}, A)$

Figure 8 Executing ghost operations. The first three exhibit the same structure: 1. `exhale_ext` determines the footprint of the operation and transfers it from $\overline{\sigma_i} \cdot \sigma_{ops}$ into σ_{emp}, yielding $\overline{\sigma'_i} \cdot \sigma'_{ops}$ and σ'_{used}; 2. the actual operation is performed, rewriting σ'_{used} into σ''_{used}; 3. the execution continues in updated states. The packaging ghost operation proceeds analogous to the package statement. The last case, $\texttt{exec}(_, A)$, handles assertions with no further ghost operations.

For example, suppose that σ holds permission to fields `x.f` and `y.f`, with values 1, respectively, 2. The operation `package (acc(x.f) * x.f = 2) -* (acc(x.f) * acc(y.f) * x.f = y.f)` will succeed, and the footprint `acc(y.f)` will be removed from the current state. The fact `y.f = 2` from the original state will be used for checking the wand's RHS (which is justified, since the permission $\texttt{acc}(y.f)$ is taken from the current state for the

wand's footprint). But importantly, the fact `x.f = 1` (which contradicts the wand's LHS) will *not* be used when checking the wand's RHS, since permission to this location is not taken from the current state. For this reason, packaging the alternative wand `package (acc(x.f) * x.f = 2) -* (acc(x.f) * acc(y.f) * false)` will fail, as it should.

For simplicity, our presentation in Figure 7 does not include cases for handling permissions under conditionals, i.e. when wands such as `true -* (b ? acc(x.f) : acc(x.g))` are to be packaged. Effectively, the footprint calculation must *branch* on the condition b, finding an appropriate footprint for when b is true, and another for when b is false. The tool can then either overapproximate, by removing at least as much as is taken in each branch, or (as in our implementation), can record the removal of the footprint information as being conditional on the value of b.

For example, assume that the current state satisfies `acc(x.f) * acc(x.g)`. Packaging an instance of the wand above succeeds, resulting in a state satisfying `(!b ? acc(x.f) : acc(x.g)) * (true -* (b ? acc(x.f) : acc(x.g)))`.
Consequently, trying to `assert acc(x.f)` will fail in that state (and likewise for `acc(x.g)`), but `assert (b ⇒ acc(x.g))` will succeed. Moreover, after applying the wand instance, `assert acc(x.f) * acc(x.g)` will succeed. We illustrate this handling of conditionals in one of our online examples (cf. Section 6).

4 Integrating Ghost Operations

The magic wand support described in the previous section forms the core of our solution, but it is not yet expressive enough to integrate well with features of the full logic such as predicates. In particular, in the proof (packaging) of a new magic wand instance, it is often necessary to be able to specify ghost operations *between* the hypothetical addition of the wand's LHS, and the proof of the RHS, for example, because the wand's RHS is a predicate instance that can only be folded once the state described by the wand's LHS is provided.

Our running example (Figure 2) exhibits an instance of this situation on line 19, when re-establishing the magic wand in the loop invariant. Recall that the wand `(xs ≠ null ⇒ List(xs)) -* List(ys)` expresses that we can obtain a predicate instance describing the complete list if we give up the "remainder list" starting from `xs`. Consider how we can re-establish this invariant at the end of the loop body: in particular, the state at line 18. In this state, we have permissions to the fields `zs.val` and `zs.next` of the current node (obtained from the `unfold` at line 15). We also have the magic wand instance `w` from the loop invariant at the beginning of the iteration (line 12), which has the same RHS, but requires `List(zs)` on its LHS. We don't directly hold enough permissions to package the wand instance needed in our new loop invariant; conceptually, those missing are in the footprint of the wand instance `w`. However, given the LHS assertion `(xs ≠ null ⇒ List(xs))`, we *can* obtain the RHS if we first fold the predicate instance `List(zs)`, and then apply the wand instance `w`. These ghost operations explain *how*, given the LHS, we can rearrange the permissions in our state to obtain the desired RHS, which requires in the process the additional permissions `acc(zs.val)` and `acc(zs.next)` and the wand instance `w` (these constitute the footprint of the new wand instance).

In order to allow such ghost operations to be expressed when packaging wand instances, we generalise the `package` statement to the form `package` $A \mathrel{-\!\!*} G$, where G is an assertion A possibly nested inside ghost operations, as defined by:

$$G \;::=\; A \;\mid\; \texttt{folding}\,P(e)\,\texttt{in}\,G \;\mid\; \texttt{unfolding}\,P(e)\,\texttt{in}\,G \;\mid\; \texttt{packaging}\,A \mathrel{-\!\!*} G\,\texttt{in}\,G \;\mid\; \texttt{applying}\,A \mathrel{-\!\!*} A\,\texttt{in}\,G$$

Note that we include syntax for nesting an assertion inside a ghost operation for each ghost operation that we support in statement position (`(un)fold`, `package`, `apply`; in other verifiers, more could be added). The difference is that the syntax here indicates that the ghost operation should be applied *during* the footprint computation for the new wand instance, rather than in the current state.

A successful `package` $A \mathrel{-\!\!*} G$ operation does *not* add a wand instance of the form $A \mathrel{-\!\!*} G$ to the state (which is not an assertion according to Section 2.3), but rather $A \mathrel{-\!\!*} A'$ where A' is the assertion nested in the ghost operations. The role of the ghost operations is to indicate only *how* the wand's semantics can be achieved, but they do not affect *what* the resulting wand instance represents. We write `nested`(G) to denote the assertion nested inside the ghost operations. For example, `nested(folding List(zs) in applying w in List(xs))` = `List(xs)`.

Our automatic footprint computation is extended to support these ghost operations, as shown in the rules in Figure 8. These rules specify a modified definition of `package`, as well as rules for executing the ghost operations. Such execution requires finding and transferring suitable permissions from the (stack of) input states, such that the specified ghost operation can be executed; for example, in order to execute a `folding`, we must find the permissions required by the body of the corresponding predicate instance. The state *resulting* from the successfully-executed ghost operations is maintained as the separate parameter σ_{ops} (which is initially empty apart from assumptions about local variables). In effect, each ghost operation rewrites already existing permissions into a different representation, which is then available to subsequent ghost operations or the eventual call to `exhale_ext`, once all ghost operations have been handled.

As an example, consider the `package` statement on line 19 of Figure 2, and assume that σ denotes the state before the `package` statement. Hence, line 19 corresponds to performing an operation σ.`package`($A \mathrel{-\!\!*}$ `folding List(zs) in` ...). Following Figure 7, this will result in `exec`($\sigma \cdot \sigma_A, \sigma_{emp}$, `folding List(zs) in` ...), which means that all permissions necessary for executing the ghost operations must come from either the current state σ or the hypothetical LHS state σ_A.

The rules for executing the first three ghost operations shown in Figure 8 (`(un)folding` and `applying`) exhibit the same structure: First, `exhale_ext` is used to find the necessary state for executing the ghost operation at hand (including checking that all necessary boolean assertions are true), and to transfer that state from $\overline{\sigma_i} \cdot \sigma_{ops}$ into σ_{emp}, which yields $\overline{\sigma_i'} \cdot \sigma_{ops}'$ (the remainders of the input states) and σ_{used}'. Next, the actual ghost operation is performed on σ_{used}', which is thereby rewritten into σ_{used}''. Note that this operation is guaranteed to succeed because of the preceding `exhale_ext`. This rewriting of the state does not change which assertions are satisfied by the state in terms of the ideal (equirecursive) semantics of assertions, but for a verifier which differentiates between predicate instances and their bodies (and between wand instances and their footprints), the ghost operation can affect what the tool can show about the resulting state. Finally, the execution continues in the updated states.

In the context of the operations in line 19 of our running example, the execution of the ghost operation `folding List(zs) in` ... proceeds by invoking `exhale_ext`($\sigma \cdot \sigma_A \cdot \sigma_{emp}, \sigma_{emp}$, `Body(List(zs))`), which transfers the footprint of the body of `List(zs)` (see Figure 1) to σ_{emp} (the second argument). The footprint comprises the permissions corresponding to `acc(zs.val)` and `acc(zs.next)`, and, assuming `zs` $\neq$ *null*, the predicate instance `List(zs)`. The algorithm tries to take as many permissions as possible from the state on top of the stack (σ_A in our example), but since σ_A only provides `List(zs.next)`,

the other permissions are taken from σ. The resulting stack of states is $\sigma' \cdot \sigma'_A \cdot \sigma_{emp}$ (where σ'_A is also essentially σ_{emp}), along with the single state σ'_{used} which is a state sufficient to satisfy `Body(List(zs))`, i.e. $\sigma'_{used} \vDash$ `Body(List(zs))`.

In the next step, σ'_{used} is rewritten into σ''_{used} by folding `List(zs)`, which replaces the footprint of the predicate body by an instance of the predicate[8].

Finally, the execution of the package statement from line 19 continues by invoking `exec(`$\sigma' \cdot \sigma'_A, \sigma_{emp} \uplus \sigma''_{used}$, `applying w in List(ys))`: executing the `applying` ghost operation proceeds by transferring `List(zs)` (the LHS of `w`) and the wand instance `w` itself from $\sigma' \cdot \sigma'_A \cdot \sigma''_{used}$to another fresh σ_{emp}. In particular, `List(zs)` is taken from σ''_{used}, and `w` is taken from σ'. Afterwards, `apply` is used to rewrite the state that now contains `List(zs)` and `w` such that it contains the RHS of `w`, i.e. `List(ys)`. Finally, `List(ys)` itself is transferred into a fresh σ_{emp} (see the last rule of Figure 8). The execution returns to the `package` rule from the start of the figure, in which the σ'_{used} is essentially a state satisfying exactly `List(ys)` – and as such, satisfies the RHS of the wand instance we set off to package. The remainder of the LHS state, σ'_A, is discarded (it is essentially empty in our case anyway), and the remainder of the current state, σ', is extended with an instance of $A \mathrel{-\!\!*}$ `List(ys)` before it is used for the verification of the rest of the program. The permissions taken from σ' compared with σ (i.e. `acc(zs.val)` and `acc(zs.next)`, as well as the wand instance `w`) conceptually belong to the newly-added wand's footprint.

Two cases from Figure 8 remain to explain: The ghost operation `packaging` $A \mathrel{-\!\!*} G$ (representing a recursive packaging of a wand instance, necessary for example for packaging nested magic wands of the form $A \mathrel{-\!\!*} (B \mathrel{-\!\!*} C)$) works similarly to the `package` statement, i.e. it creates an extra LHS state σ_A satisfying A, pushes it onto the stack of already existing states, and executes ghost operations potentially occurring in G (to which σ_A is available). Finally, it adds the magic wand $A \mathrel{-\!\!*}$ `nested`(G) to the state, and continues the execution.

The last case, executing a ghost-operation-free assertion A, only applies to the inner-most assertion nested inside a chain of ghost operations (i.e. A is `nested`(G) for some G). At this point, the footprint computation falls back to the `exhale_ext` algorithm of Figure 7.

Note that automatic footprint computation in the presence of ghost operations is especially challenging in the case of nested package operations; one has to track the various hypothetical states carefully, and ideally remove permissions from these states preferentially, since any removal from the original state σ can affect the further verification of the rest of the program. The precise details of the algorithms are quite complex, but specifying them at the level of abstraction shown here helped to guide our original implementation and avoid soundness issues. We give a soundness argument in the following subsection.

4.1 Soundness of the Footprint Computation

In this subsection we sketch a soundness argument for the core of our magic wand support: the soundness of our approach depends essentially on the correctness of our footprint computation. We focus on showing that the state removed by a package operation satisfies the properties that: it was a part of the original state, and it satisfies the semantics of the newly-packaged wand (thus, any future `apply` of the wand instance will be justified). We formulate similar results for each of our `transfer`, `exhale_ext` and `exec` definitions, which we then instantiate to show the desired property for `package`. By convention, we use unprimed σ variables

[8] One way of implementing σ'_{used}.`fold`$(\cdot)$ is to exhale the predicate body and inhale the predicate instance. However, some verifiers might choose to implement folding predicate instances in a custom way which preserves extra information [11]; we leave the precise implementation up to the particular verifier.

to denote input states, primed versions σ' to indicate corresponding output states, and $\widehat{\sigma}$ versions to indicate the "removed parts" of the input state. We also use an "inclusion" relation $\sqsubseteq$ on states, which has the meaning: $\sigma \sqsubseteq \sigma'$ iff $\forall A.(\sigma \vDash A \Rightarrow \sigma' \vDash A)$.

▶ **Theorem 1.** *If* $\mathit{transfer}(\overline{\sigma_i}, \sigma_{used}, \mathtt{acc}(e.f))$ *succeeds with result* $(\overline{\sigma_i'}, \sigma'_{used})$*, then there exist* $\overline{\widehat{\sigma_i}}$*, such that:*

(P_1) $\overline{\sigma_i' \uplus \widehat{\sigma_i} \ \sqsubseteq\ \sigma_i}$

(P_2) $\sigma'_{used} \ \sqsubseteq\ (\overline{\uplus \widehat{\sigma_i}}) \uplus \sigma_{used}$

(P_3) $\forall A.(\sigma_{used} \vDash A \Rightarrow \sigma'_{used} \vDash A * \mathtt{acc}(e.f))$

The first two properties of Theorem 1 state that `transfer` does not add permissions (and assumptions) when it rewrites $(\overline{\sigma_i}, \sigma_{used})$ into $(\overline{\sigma_i'}, \sigma'_{used})$. Essentially, the original states σ_i are each split into a $\widehat{\sigma_i}$ part (which is removed from the state and makes up part of σ'_{used}) and a remaining σ_i'. The output σ'_{used} consists of these removed parts, plus anything that was originally in σ_{used}. The third property of Theorem 1 essentially expresses that it is the *extra* parts of σ'_{used} that satisfy $\mathtt{acc}(e.f)$). In particular, repeated (successful) calls to `transfer` build up a state which satisfies the conjunction of all transferred permissions.

Proof of Theorem 1. The proof proceeds by induction on the input stack. In case of the empty stack ϵ, `transfer` fails, which contradicts the assumption made in Theorem 1. In case of a non-empty input stack $\overline{\sigma_i} \cdot \sigma$, let us consider the if-branch first: $\overline{\sigma_i}$ remains unchanged, hence, we choose $\overline{\widehat{\sigma_i}}$ to be a stack of empty states $\varnothing$. In addition, let $\widehat{\sigma}$ be $\varnothing.\mathtt{addAcc}(v, f)$. Given these choices, $\overline{\widehat{\sigma_i}} \cdot \widehat{\sigma}$ satisfies (P_1) and (P_2). The output state σ''_{used} is the input state σ_{used} with exactly one extra permission added (along with any assumptions about that location's value), corresponding to $\mathtt{acc}(e.f)$. Thus, (P_3) also holds.

In the else-branch, the induction hypothesis yields appropriate $\overline{\widehat{\sigma_i}}$ for all states but the last; choosing $\widehat{\sigma}$ to be $\varnothing$ yields the desired properties. ◀

▶ **Theorem 2.** *If* $\mathit{exhale_ext}(\overline{\sigma_i}, \sigma_{used}, A)$ *succeeds with result* $(\overline{\sigma_i'}, \sigma'_{used})$*, then there exist* $\overline{\widehat{\sigma_i}}$*, such that* (P_1) *and* (P_2) *from Theorem 1 hold. Moreover:*
(P_3) $\forall A'.(\sigma_{used} \vDash A' \Rightarrow \sigma'_{used} \vDash A' * A)$ *holds.*

Proof of Theorem 2. The proof proceeds by induction on the structure of the assertion A. In case of $\mathtt{acc}(e.f)$, the desired results follow directly from `transfer`; in case of e, it suffices to choose $\overline{\widehat{\sigma_i}}$ to be a stack of empty states. The only interesting case is that of $A_1 * A_2$: Let $\overline{\widehat{\sigma_i}'}$ and $\overline{\widehat{\sigma_i}''}$ be the states whose existence follows from applying the induction hypothesis to the first and second recursive invocations of `exhale_ext`, respectively. Choosing $\overline{\widehat{\sigma_i}}$ to be $\overline{\widehat{\sigma_i}' \uplus \widehat{\sigma_i}''}$, it is straight-forward to combine the assumptions about $\overline{\widehat{\sigma_i}'}$ and $\overline{\widehat{\sigma_i}''}$ to show that (P_1), (P_2) and (P_3) hold for $\overline{\widehat{\sigma_i}}$. ◀

Theorem 2 is essentially a generalisation of Theorem 1: the first two properties of Theorem 2 are identical to those of Theorem 1; the third differs since the algorithm finds suitable substates to satisfy a general assertion A, instead of only a single required permission. We can use this result to reason about `exec`:

▶ **Theorem 3.** *If* $\mathit{exec}(\overline{\sigma_i}, \sigma_{ops}, G)$ *succeeds with result* $(\overline{\sigma_i'}, \sigma'_{ops})$*, then there exist* $\overline{\widehat{\sigma_i}}$*, such that:*

(P_1) $\overline{\sigma_i' \uplus \widehat{\sigma_i} \ \sqsubseteq\ \sigma_i}$

(P_2) $\sigma'_{ops} \ \sqsubseteq\ (\overline{\uplus \widehat{\sigma_i}'}) \uplus \sigma_{ops}$

(P_3) $\sigma'_{ops} \vDash \mathit{nested}(G)$

Proof of Theorem 3. The proof proceeds by induction on the structure of the (ghost operations in) the assertion G. The cases of `folding`, `unfolding` and `applying` are all similar to each other: From Theorem 2 and from the induction hypothesis applied to `exec`, the existence of $\overline{\widehat{\sigma_i}'}$ and $\overline{\widehat{\sigma_i}''}$ satisfying the appropriate conditions follows. Choosing $\overline{\widehat{\sigma_i}}$ to be $\overline{\widehat{\sigma_i}' \uplus \widehat{\sigma_i}''}$, it suffices to combine the assumptions about $\overline{\widehat{\sigma_i}'}$ and $\overline{\widehat{\sigma_i}''}$, as well as to observe that the particular ghost operation applied does not (with respect to the semantics of the logic) increase the true assertions in the resulting state, to show that P_1 and P_2 hold for $\overline{\widehat{\sigma_i}}$. Since σ'_{ops} (the state returned) is the second component of the result of the recursive invocation of `exec`, the assumptions gained for this call from the induction hypothesis suffice to show that (P_3) holds. The case of `packaging` is similar to the above cases, but showing P_2 is more involved (similarly to our argument about `package` in Section 4.1): it is necessary to observe that: (i) the call to $\texttt{exec}(\ldots, G_1)$ removes a suitable footprint of $A \mathrel{-\!\!*} G_1$ from $\overline{\sigma_i} \cdot \sigma_{ops} \cdot \sigma_{emp}$ (by similar argument to that for `package` in Section 4.1), and (ii) that any footprint of $A \mathrel{-\!\!*} G_1$ is at least as expressive (in terms of which assertions can be deduced from it) as the wand $A \mathrel{-\!\!*} \texttt{nested}(G_1)$ which is added to the state in its place. In case of an assertion A without ghost operations, all properties follow directly from Theorem 2 applied to the call of `exhale_ext`. ◀

In Theorem 3, the first two properties are similar to those for our other results, essentially expressing that we remove parts of the input states to obtain σ'_{ops}. The third property differs; in general, executing the ghost operations involves rewriting the original state σ_{ops} (and pulling in extra parts of the stack of input states which are found to be missing). This is different from `exhale_ext`, which seeks to *add* everything in the required assertion to the pre-existing σ_{used}, which gives us the stronger third property in Theorem 1 and Theorem 2.

Nonetheless, if we consider the definition of `package` in Figure 8, we can see that these properties are sufficient to guarantee that the call to `exec` removes a correct footprint. In particular, instantiating Theorem 3 for this call, we obtain that there exist $\widehat{\sigma}, \widehat{\sigma_A}$ such that $(\sigma' \uplus \widehat{\sigma}) \sqsubseteq \sigma$, and, (combining the three properties from the theorem) $\widehat{\sigma} \uplus \widehat{\sigma_A} \vDash \texttt{nested}(G)$ (note that σ_{ops} is σ_{emp} for this call). From this, we obtain $\widehat{\sigma} \uplus \sigma_A \vDash \texttt{nested}(G)$. Since σ_A is an arbitrary state satisfying A, $\widehat{\sigma}$ is an appropriate footprint state for the wand. This state is removed from σ (resulting in the returned σ'), thus (assuming our algorithm reaches this point without failure) adding the wand instance in its place is justified, according to its semantics.

5 Inferring Annotations

In order to reduce the annotation overhead involved in specifying magic-wand-related ghost operations, we have extended the approach presented so far with a set of simple (and optional) heuristics, which attempt to insert additional `package` and `apply` operations into an input program. As described in Section 4, a `package` operation may also require nested ghost operations in order to succeed; our heuristics also attempt to infer these appropriately.

Our heuristics are *failure-directed*: if exhaling an assertion A, (e.g. a loop invariant), fails due to insufficient permissions (to a field, a predicate or a wand), then we apply the heuristics to search for ghost operations that avoid the failure. The heuristics search (in a depth-first manner) for a sequence of ghost operations that would rewrite the state such that the initially missing permissions can be found. The width of the search tree is bounded by the number of predicate and magic wand instances held in the current state (which is finite and typically small). The depth of the search is bounded by a configurable threshold. We also order the candidate ghost operations according to a number of syntactic criteria on the

```
0   var xs := ys;
1   sum := 0
2
3   define A
4   define B List(ys)
5
6   while (xs ≠ null)
7     invariant (xs ≠ null ⇒ List(xs)) *
8       ((xs ≠ null ⇒ List(xs)) -* List(ys)) *
9       sum = old(sum_rec(ys)) - (xs = null ? 0 : sum_rec(xs))
10  {
11    unfold List(xs)
12    sum := sum + xs.val
13    xs := xs.next;
14  }
```

Figure 9 Verified version of the body of `sum_rec` (Figure 1), with heuristics enabled.

symbolic state and program text, as a coarse estimate of which operations are "likely" to be successful, for example by preferentially unfolding predicate instances whose bodies appear to contain a suitable permission.

As an example, consider the loop body from our running example (Figure 2), and assume that the `package` statement in line 19 were removed, which would prevent the verifier from (immediately) showing that the invariant is preserved by the loop body. In particular, the verifier would fail to find the wand instance `A -* List(ys)` in the current state. This would trigger the heuristics, which would first detect that there is no predicate (or wand) instance in the current state that could be unfolded (or applied) to get a suitable wand instance. The heuristics would then try to package `A -* List(ys)`, which would fail because the desired predicate instance `List(ys)` would not be found either in the current state or the hypothetical state from the wand's left-hand-side. This would trigger the heuristics again, and result in an attempt to apply the wand instance `w` (which mentions `List(ys)`). Applying `w` would fail as well - because this wand's left-hand-side `List(zs)` is missing. The heuristics would be triggered once again, and try to fold `List(zs)`, which succeeds. The previously failing operations are then retried: that is, applying `w` and exhaling `List(ys)`, both of which succeed now. With these nested ghost operations, the initially triggered packaging of `A -* List(ys)` also succeeds, which enables the verifier to find the previously missing wand instance, and therefore, to show that the loop invariant is preserved.

Our heuristics allow us to remove all `package` and `apply` statements from the examples listed in Section 6. We can also remove `w` and `zs` (which were only used to facilitate writing a `package` statement) declared on line 12 and line 14, respectively, of Figure 2. The code shown in Figure 9 is verified by our implementation [29] when heuristics are enabled.

6 Implementation

Our implementation [29][9] includes the examples listed below (as well as a number of regression tests), all of which have been verified on a Intel Core i7-2600K 3.40GHz machine running Windows 7 x64 from an SSD. For each example we also include a version with the suffix `_heuristics.sil`, which is the example with heuristics activated and with all magic-wand-related annotations removed. The reported runtimes are averaged over ten runs per example

[9] In Silver, the separating conjunction is denoted by `&&`; for simplicity, we used `*` in this paper

(the standard deviations were always less than 0.1s). We state two (averaged) runtimes per example: the first figure is the overall runtime, which includes time for parsing and type-checking the example, starting-up the prover (i.e. Z3), and the verification time; the second records the verification time only.

- `list_sum.sil` is the running example from our paper. It verifies in 2.2s/0.7s, both with and without heuristics enabled (to infer the necessary magic-wand-related annotations).
- `list_insert.sil` is an encoding of an iterative algorithm for inserting a value into a sorted linked list. It verifies in 3.0s/1.5s (3.5s/2.0s with activated heuristics).
- `tree_delete_min.sil` is an encoding of challenge 3 from the VerifyThis verification competition at Formal Methods 2012, which was to verify an iterative implementation removing the minimal element from a binary search tree. The example verifies in 2.6s/1.1 (2.8s/1.4s with activated heuristics). VerCors [2] (the only comparable tool we are aware of with magic wand support) requires substantially more annotations to specify this example, and takes 6 minutes (on a comparable machine).
- `un_currying.sil` demonstrates how nested ghost operations can be used to prove the standard "currying" and "uncurrying" property of magic wands: $A * B \mathrel{-\!\!*} C \Leftrightarrow A \mathrel{-\!\!*} (B \mathrel{-\!\!*} C)$. The "$\Rightarrow$" case is especially interesting since it requires nested `packaging` operations. The example verifies in 1.6s/0.3s (both with and without heuristics).
- `conditionals.sil` illustrates and explains how our tool handles magic wands where the footprint is affected by conditionals whose guards depend on locations that are provided by the LHS of the wand. It verifies in 1.8s/0.4s. (We do not provide a version with activated heuristics because adding assertions that trigger packaging and applying the involved wands turned out to be more overhead than explicitly packaging/applying them.)

7 Conclusions and Related Work

We have presented a novel technique enabling the support of magic wands in automatic verification tools. Our approach requires moderate additional specification overhead and is still expressive enough to encode general uses of the logical connective. Most important is our ability to compute suitable footprints for magic wands *automatically*, which greatly simplifies the annotation effort required. We have implemented the described support as an extension of the verification tool Silicon [16], which supports implicit dynamic frames assertions. Our work makes few assumptions about the underlying verifier and specific logic, and should be easy to apply in other tools, such as verifiers for separation logics.

Lee and Park have recently developed a proof system for a separation logic supporting the magic wand connective [19], which also provides a decision procedure for propositional separation logic (i.e. without variables). In a richer logic such as ours, however, the magic wand is known to be undecidable [5]. Our work addresses this difficulty with the combination of `apply` and `package` annotations (which can often be inferred by our heuristics), along with novel algorithms for computing appropriate magic wand footprints automatically.

In parallel with our work, Blom and Huisman [2] have developed support for magic wands in their VerCors verifier. VerCors translates Java programs with separation-logic-style specifications into Chalice programs [21], and magic wands are eliminated during the translation by a clever encoding into additional Chalice classes whose instances ("witness objects") represent magic wand instances. This translation is automatic, but similar annotations to our approach are needed to direct the creation and use of magic wands. In contrast to our approach, the user must also manually specify annotations defining the permissions and logical facts to be used from the current state for each wand's footprint, which are then

combined to show the wand's RHS via arbitrary user-defined ghost code. The ability to use arbitrary code is potentially more flexible than the ghost operations our tool supports (for example, ghost methods could be employed), but the resulting annotation overhead is significantly higher than with the automatic footprint computation presented in this paper (even comparing without the additional heuristics described in Section 5). Moreover, their translation does not support nested wands such as $A \mathrel{-\!\!*} (B \mathrel{-\!\!*} C)$ or wands inside predicate definitions (although we believe it could be extended to handle the latter).

In the context of a permission-based type system, Boyland [4] has defined a "sceptre" operator to represent "borrowing" of permission. This connective is more restricted than the general magic wand, but is sufficient for many loop invariants, such as the one in our example. The PhD thesis of Retert [28] provides an abstract-interpretation-based approach supporting this connective.

The specific problem of rewriting and maintaining appropriate predicate definitions during data structure traversals has already received much attention. Without an alternative to simple `fold`/`unfold` annotations, one needs to define a new predicate type to represent "partial" versions of the data structure, and write ghost methods to "append" to this partial version, as well as to rewrite it into the original predicate once the traversal is completed. The problems of tracking suitable permissions in loop invariants are discussed in detail by Tuerk [34], who proposed alternative pre/postcondition specifications for loops. A magic wand of the form: $pre_{rest} * (post_{rest} \mathrel{-\!\!*} post_{all})$ gives an alternate expression of his idea (where "*rest*" refers to the remaining loop iteration, and "*all*" the entire loop). Making use of magic wand support is more general than Tuerk's proof rule, for example when further code after the loop is needed before restoring the overall predicate, as in the tree-min-delete challenge (Section 6).

A variety of existing work aims to reduce the annotation overhead associated with managing and rewriting predicate definitions with explicit `fold` and `unfold` operations. For example, Smallfoot [1] and Grasshopper [27] achieve concise specifications without user direction by building in specific support for list and tree predicates. Lee et al. [18] provide a static analysis capable of identifying when objects participate in many such data structures simultaneously. Nguyen and Chin [23] and Brotherston et al. [6] provide techniques for proving and applying user-supplied lemmas automatically. Chin et al. [7] provide support for a wider class of predicate definitions, including functional abstractions of data structures, provided that one reference parameter is traversed in the predicate's definition. Their entailment checker "carves out" a suitable portion of the input state, which (for one input state) is similar to the operation of our footprint computation algorithm.

These techniques improve the usability of recursive predicate reasoning, and can complement our work in a practical tool. Each comes with limitations: they cannot be applied equally to fully general predicates. One consequence of available magic wand support is that iterative code (such as our running example) can be specified without the need for extra predicate types to represent "partial" versions of data structures. These extra predicates do not describe structures which the program operates on, and are cumbersome to define for structures more complex than linked lists; loop invariants employing magic wands can be defined analogously for other data structures, and also support the specification of functional properties (e.g. the use of `sum_rec` in our example).

VeriFast [14] is a mature and expressive verifier for programs annotated with separation logic. We believe it is possible to partly work around the absence of magic wands using *lemma function pointers* and predicates. One can encode a wand $A \mathrel{-\!\!*} B$, using a predicate F (representing the wand's footprint), and a pointer to a lemma function with precondition

$F * A$ and postcondition B, whose body shows how to rewrite the state. The need to define the footprint manually, however, entails substantial additional overhead (to define a predicate for each footprint, and the appropriate lemma methods for manipulating them) for the user compared with our technique of automatic footprint computation.

As future work, we are interested to investigate other applications of our magic wand support, such as reasoning about *closures*, for which it is useful to be able to reason about connecting calls of closures together without knowing their specifications concretely. We are also interested in encoding existing by-hand proofs using our prototype implementation, e.g. parts of the proofs from [10, 9, 12]. The developers of the Viper verification tools also plan to incorporate our magic wand support into their tool infrastructure.

Acknowledgements. The authors were funded by the Hasler Foundation. We thank the anonymous referees for suggesting many improvements to the explanation of our work. We are also very grateful to Peter Müller, Ioannis T. Kassios, Uri Juhasz and John Boyland for extensive feedback on earlier versions of this paper.

References

1. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.
2. Stefan Blom and Marieke Huisman. Witnessing the elimination of magic wands. Technical Report TR-CTIT-13-22, University of Twente, November 2013.
3. John Tang Boyland. Checking interference with fractional permissions. In *SAS*, 2003.
4. John Tang Boyland. Semantics of fractional permissions with nesting. *ACM Trans. Program. Lang. Syst.*, 32(6):22:1–22:33, August 2010.
5. Rémi Brochenin, Stéphane Demri, and Etienne Lozes. On the almighty wand. *Journal of Information and Computation*, 211:106–137, February 2012.
6. James Brotherston, Dino Distefano, and Rasmus Lerchedahl Petersen. Automated cyclic entailment proofs in separation logic. In *CADE'11*, pages 131–146, 2011.
7. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006–1036, 2012.
8. Dino Distefano and Matthew J. Parkinson. jStar: towards practical verification for java. In *OOPSLA*, pages 213–226, 2008.
9. Mike Dodds, Suresh Jagannathan, and Matthew J. Parkinson. Modular reasoning for deterministic parallelism. In *POPL*, pages 259–270, 2011.
10. Christian Haack and Clément Hurlin. Resource usage protocols for iterators. *Journal of Object Technology*, 8:55–83, 2009.
11. Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In *ECOOP*, pages 451–476, 2013.
12. Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. In *POPL'13*, pages 523–536. ACM, 2013.
13. Zhe Hou, Ranald Clouston, Rajeev Goré, and Alwen Tiu. Proof search for propositional abstract separation logics via labelled sequents. In *POPL*, pages 465–476, 2014.
14. Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical report, Katholieke Universiteit Leuven, August 2008.
15. Jonas Braband Jensen, Lars Birkedal, and Peter Sestoft. Modular verification of linked lists with views via separation logic. *Journal of Object Technology*, 10:2:1–20, 2011.

16 U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. Technical report, ETH Zurich, 2014.
17 Neelakantan R. Krishnaswami. Reasoning about iterators with separation logic. In *SVCBS*, pages 83–86, New York, NY, USA, 2006. ACM.
18 Oukseh Lee, Hongseok Yang, and Rasmus Petersen. Program analysis for overlaid data structures. In *CAV*, volume 6806 of *LNCS*, pages 592–608. Springer Berlin Heidelberg, 2011.
19 Wonyeol Lee and Sungwoo Park. A proof system for separation logic with magic wand. In *POPL*, pages 477–490, 2014.
20 K. Rustan M. Leino. This is Boogie 2. Available from `http://research.microsoft.com/en-us/um/people/leino/papers.html`.
21 K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In *ESOP*, pages 378–393, 2009.
22 Toshiyuki Maeda, Haruki Sato, and Akinori Yonezawa. Extended alias type system using separating implication. In *TLDI*, pages 29–42. ACM, 2011.
23 Huu Hai Nguyen and Wei-Ngan Chin. Enhancing program verification with lemmas. In *CAV*, volume 5123 of *LNCS*, pages 355–369. Springer Berlin Heidelberg, 2008.
24 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, London, UK, 2001. Springer-Verlag.
25 M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM Press, 2005.
26 M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *In LMCS*, 8(3:01):1–54, 2012.
27 Ruzica Piskac, Thomas Wies, and Damien Zufferey. GRASShopper – complete heap verification with mixed specifications. In *TACAS*, pages 124–139. Springer, 2014.
28 William S. Retert. *Implementing Permission Analysis*. PhD thesis, University of Wisconsin at Milwaukee, Milwaukee, WI, USA, 2009.
29 Malte Schwerhoff and Alexander J. Summers. Implementation. Available from `www.pm.inf.ethz.ch/research/viper.html`.
30 Jan Smans. *Specification and Automatic Verification of Frame Properties for Java-like Programs*. PhD thesis, FWO-Vlaanderen, May 2009.
31 Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, pages 148–172, 2009.
32 Jan Smans, Bart Jacobs, and Frank Piessens. Heap-dependent expressions in separation logic. In *FMOODS/FORTE*, pages 170–185, 2010.
33 Alexander J. Summers and Sophia Drossopoulou. A formal semantics for isorecursive and equirecursive state abstractions. In *ECOOP*, pages 129–153, 2013.
34 Thomas Tuerk. Local reasoning about while-loops. In *VSTTE*, 2010.
35 Hongseok Yang. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In *SPACE*, 2001.

Modular Verification of Finite Blocking in Non-terminating Programs

Pontus Boström[1] and Peter Müller[2]

1 **Åbo Akademi University, Finland,** pontus.bostrom@abo.fi
2 **Department of Computer Science, ETH Zurich, Switzerland,** peter.mueller@inf.ethz.ch

Abstract

Most multi-threaded programs synchronize threads via blocking operations such as acquiring locks or joining other threads. An important correctness property of such programs is for each thread to make progress, that is, not to be blocked forever. For programs in which all threads terminate, progress essentially follows from deadlock freedom. However, for the common case that a program contains non-terminating threads such as servers or actors, deadlock freedom is not sufficient. For instance, a thread may be blocked forever by a non-terminating thread if it attempts to join that thread or to acquire a lock held by that thread.

In this paper, we present a verification technique for finite blocking in non-terminating programs. The key idea is to track explicitly whether a thread has an obligation to perform an operation that unblocks another thread, for instance, an obligation to release a lock or to terminate. Each obligation is associated with a measure to ensure that it is fulfilled within finitely many steps. Obligations may be used in specifications, which makes verification modular. We formalize our technique via an encoding into Boogie, which treats different kinds of obligations uniformly. It subsumes termination checking as a special case.

1998 ACM Subject Classification D.2.4 Software/Program Verification, D.1.3 Concurrent Programming

Keywords and phrases Program verification, concurrency, liveness, progress, obligations

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.639

1 Introduction

Most multi-threaded programs synchronize threads via blocking operations such as acquiring locks, receiving messages on a channel, awaiting conditions, or joining other threads. The correctness of such programs typically relies on all threads being able to make progress, that is, not being blocked forever. For instance, a producer-consumer system typically requires that each producer will eventually succeed in acquiring the lock to a shared buffer. Existing work [10, 15] has demonstrated that for *terminating* programs, progress can be ensured by (1) avoiding starvation through fair scheduling and (2) showing that the program does not create circular situations akin to deadlock, where each thread on a cycle waits for the next thread to perform an action to unblock it.

However, this solution is insufficient if programs contain potentially non-terminating threads such as actors, servers, watch-dogs, etc. Such threads potentially defer the execution of an unblocking operations forever. For instance, a thread may be blocked forever by a non-terminating thread if it attempts to join that thread or to acquire a lock held by that thread.

© Pontus Boström and Peter Müller;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 639–663

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper, we present a verification technique for finite blocking in non-terminating programs. The key idea is to track explicitly whether a thread has an obligation to perform an operation that unblocks another thread. For instance, a thread may receive on a channel only if another thread has an obligation to send a message on that channel, and a thread may join another thread only if the latter has an obligation to terminate. To handle non-termination, we associate each obligation with a measure (also called variant or ranking function) and check that each thread satisfies its obligations within finitely many steps, even if the thread does not terminate. Our verification technique guarantees *finite blocking* for programs with a finite number of threads in each state and fair scheduling. That is, each thread in an execution of a verified program either terminates or runs forever, but no thread is blocked forever.

Even though the finite blocking guarantee relies on fairness, our technique is also useful for non-fair systems. First, proving that no thread postpones unblocking another thread indefinitely is still necessary (although not sufficient without fairness) for progress; a violation of this property is an error. Second, although this paper focuses on finite blocking, the concept of obligations is more general and can be used to specify and verify other liveness properties for both sequential and concurrent programs, for instance, that each asynchronous task will be awaited or that a given I/O operation will be performed eventually.

Our verification technique is modular, that is, verifies each method independently, without knowledge of the program context in which it is used and the threads executing concurrently. We formalize the technique for a language without heap memory, but the style of reasoning integrates well with permission logics such as separation logic [22] and implicit dynamic frames [23], and can be automated in a similar way. In particular, our technique produces verification conditions that are amenable to automation using SMT-solvers. We have manually encoded several challenging examples and verified them successfully in Boogie [1]. These examples include producer-consumer communicating over a channel, bi-directional channels, and parallel binary tree processing; they exercise all major features of our approach.

Contributions and Outline. This paper makes the following contributions:

1. It presents the first modular verification technique for finite blocking in non-terminating programs.
2. It introduces explicit obligations with measures to uniformly specify guarantee properties [16] and verify them in standard program logics.
3. It unifies verification tasks such as proving termination, deadlock freedom, and finite blocking in one coherent methodology.
4. It adopts ideas from the Chalice verifier [15], but encodes them in a simpler way and fixes a soundness problem.

We give an informal overview of our verification technique in Sec. 2 and introduce the programming and assertion language in Sec. 3. We present the encoding of assertions in Sec. 4 and of statements in Sec. 5. Sec. 6 provides an informal soundness argument. We discuss related work in Sec. 7 and conclude in Sec. 8. App. A illustrates the treatment of message passing and deadlock freedom that we adopted from Chalice.

2 Verification Technique

This section presents the main ideas of our verification technique informally.

2.1 Obligations

An obligation is associated with a thread and specifies an action that this thread must eventually perform, either itself or by delegating it to another thread. The action could be executing a certain statement, establishing certain conditions, or reaching certain program points. Since this paper focuses on the verification of finite blocking, we use obligations to enforce actions that a thread must perform to unblock another thread. We introduce a different kind of obligation for each blocking operation. For instance, a releases-obligation indicates that a thread must release a given lock to unblock a thread possibly trying to acquire it, and a terminates-obligation indicates that a thread must terminate to unblock a thread possibly trying to join it.

The obligations for different blocking operations have different characteristics along three dimensions:

1. Some obligations can be accumulated (for instance, to express that several messages must be sent on a channel or that a re-entrant lock must be released several times), whereas others cannot (for instance, an obligation to terminate).
2. For some obligations, there is a dual concept of *credit*, which expresses the permission to execute a blocking operation. We view credits as negative obligations. In particular, creating a credit creates also the corresponding obligation. Credits are necessary for those blocking operations where the very first execution will block. For instance, if channels are initially empty then receiving on a channel requires a credit to ensure that some thread has the obligation to send a message eventually. In contrast, acquiring a lock does not require a credit because the very first acquire for each lock always succeeds; each acquire then creates a releases-obligation to ensure that subsequent acquires also succeed eventually.
3. Some obligations may be delegated to other threads (for instance, an obligation to send a message), whereas others may not (for instance, obligations to terminate or to release a lock).

Despite these different characteristics, our verification technique treats obligations uniformly. To enable modular verification, we track the obligations held by the current thread on the level of individual method executions rather than the entire thread. Obligations may be passed between different method executions when a method is called, when a method terminates, and when a method is forked in a different thread (but not upon thread-join, as we will discuss later). Which obligations get transferred is expressed in the method specifications, analogously to the transfer of access permissions in implicit dynamic frames [14, 23]. For each kind of obligation, we provide an assertion that can be used in method pre and postconditions. When a method is called (or forked), the obligations required in the method precondition are transferred from the caller to the callee; analogously, the obligations provided by the method postcondition are transferred from the method to its caller upon termination. Loops are treated analogously: we track obligations per loop iteration, and the loop invariant specifies the permissions required and provided by a loop iteration.

Proof rules ensure that each obligation is held by an active method execution (an execution on the stack of any thread) or loop invariant until it is satisfied. In particular, a *leak check* ensures that when a method execution terminates, all of its remaining obligations are transferred to the caller. Moreover, *well-formedness checks* ensure that obligations cannot be lost by sending them in a message (that might never get received) or by putting them in the postcondition of a forked method (since the forked thread may never get joined). However, leaking or losing credit is permitted.

```
method A(l: Lock)
{
  acquire l;
  call R(l);
}
```

```
method R(l: Lock)
  requires releases(l);
{
  release l;
}
```

Figure 1 An example illustrating the use and transfer of obligations. We omit specifications related to concepts introduced later, in particular, obligation measures and deadlock prevention.

Fig. 1 illustrates some of the concepts introduced so far. Method `A` acquires lock `l`, thereby obtaining an obligation to release it eventually. Method `R` requires a releases-obligation to `l` in its precondition. Therefore, when `A` calls `R`, its obligation is transferred to `R`. After the call, `A` does not contain any obligations and, thus, passes the leak check. Method `R` gets rid of its obligation by releasing `l` and, thus, also passes the leak check.

2.2 Obligation Measures

Obligations allow one to track modularly which method execution is expected to perform a given unblocking operation. However, the proof rules sketched above are not sufficient to prevent a non-terminating thread from blocking another thread forever. Assume method `R` from Fig. 1 was implemented as follows:

```
method R(l: Lock)
  requires releases(l);
{
  while(true)
    invariant releases(l);
  { }
  release l;
}
```

This implementation passes the leak check since no obligations are held at the end of the method or at the end of a loop iteration after the releases-obligation has been transferred to the next loop iteration. However, the method obviously fails to release `l` because it enters a non-terminating loop before reaching the release operation.

A naïve solution would be to require that a method holds no obligations when it enters a possibly non-terminating loop or calls a possibly non-terminating method. However, this solution is too restrictive for many useful implementations. For instance, the `Await` method in Fig. 2 encodes a busy version of Java's `wait` method. The method loops until a condition `P` holds, where `P` refers to fields that are protected by a lock `l`. Therefore, `Await` will be called in states where the executing thread holds lock `l` and, hence, the method has a releases-obligation for `l`. In each loop iteration, the method releases and then re-acquires the look such that other threads may obtain the lock and establish `P`.

The naïve solution would disallow method `Await` unless one could prove that the loop will always terminate, which may be difficult in a modular setting. However, since the loop releases and re-acquires the lock `l` in each iteration, it is guaranteed not to block indefinitely any other thread that attempts to acquire `l` (assuming fair scheduling and fair locks). A similar situation occurs when a thread is expected to send an unbounded number of messages over a channel. Its send-loop might not be guaranteed to terminate, but holds a sends-obligation in each iteration (see App. A for the full example); it would therefore be rejected by the naïve solution.

```
method Await(l: Lock)
  requires releases(l, 1);
  ensures releases(l, 1);
{
  while(!P)
    invariant releases(l, 1);
  {
    release l;
    acquire l;
  }
}
```

Figure 2 A busy version of Java's `wait` method. In contrast to method `R` above, the lock is released and re-acquired in each loop iteration. We omit specifications related to deadlock prevention.

Measures. These two examples show that the naïve solution is overly conservative. It should be possible for a thread to hold obligations during a non-terminating execution as long as these obligations will be satisfied eventually. To verify this liveness property without resorting to temporal reasoning, we reduce it to a safety property by associating each obligation with a measure (also called variant or ranking function). Analogously to a termination measure, an obligation's measure is an expression that evaluates to a value in a well-founded set. Proof rules ensure that the measure is decreased in each loop iteration or recursive call, and that the obligation gets satisfied before its measure expires. This check would fail for the non-terminating version of method `R` above because there is no measure that one could choose for the releases-obligation that gets decreased during the non-terminating loop.

Fresh Obligations. Measures alone cannot distinguish between the situations in method `R` and method `Await`. In both of them, a possibly non-terminating loop holds a releases-obligation before and after each loop iteration. However, method `R` might cause indefinite blocking because the obligation is held throughout the loop body, whereas method `Await` is safe because the releases-obligation is satisfied and re-obtained in each iteration, giving other threads a chance to acquire the lock in between. To distinguish these two situations, we track explicitly whether an obligation is *fresh*, that is, has been obtained since the prestate of the current method execution or loop iteration. Fresh obligations are exempted from the check that their measure decreases before the next recursive call or loop iteration. In the `Await` method above, the measure of all releases-obligations is the constant 1, expressing that lock `l` will be released within one loop iteration. This constant measure is not decreased in the loop body. However, since acquiring the lock `l` obtains a fresh releases-obligation, it is exempted from the check that the measure decreases, and the method verifies.

Termination. Associating obligations with measures allows us to treat termination like any other obligation. Therefore, termination proofs are a special case of the general technique we propose. For instance, the factorial method in Fig. 3 promises to terminate after at most `n` recursive calls if its argument is non-negative. This termination guarantee is expressed by including a terminates-obligation with measure `n` in the method's precondition. We assume here that programmers provide the measures for termination and other obligations. Combining our technique with inference of termination measures (see Cook et al. [4] for an overview) is future work. The termination guarantee of `Fac` ensures that the join in method `Main` will not block indefinitely.

It might initially seem un-intuitive that termination as well as the satisfaction of other

```
method Fac(n: int) returns (res: int)
  requires 0 ≤ n ⇒ terminates(n);
{
  if (n ≤ 1)
    res := 1;
  else
    res := n * Fac(n - 1);
}
```

```
method Main(n: int) returns (res: int)
{
  if (0 ≤ n) {
    fork t := Fac(n);
    join res := t;
  }
}
```

Figure 3 A recursive factorial method. The terminates-obligation in the precondition expresses that the method will terminate if `n` is non-negative. The antecendent ensures that the measure of the obligation is well-founded. The main method forks a new thread to execute `Fac`. It may join this thread only because `Fac` is guaranteed to terminate and, thus, the join will not block indefinitely.

obligations is specified as a method precondition rather than a postcondition. However, this approach is consistent with the treatment of permissions in permission-based logics such as separation logic. The precondition specifies which resources get transferred from the caller to the callee. In permission logics, the transferred resources are partial heaps; here, they are obligations. So one should think of a precondition as the obligations *consumed* by the callee and of a postcondition as the obligations *provided* by the callee.

2.3 Wait Order

Finite blocking implies the absence of deadlock because each thread involved in a deadlock blocks indefinitely. A deadlock occurs if one or more threads form a cycle where each thread is blocked by its successor on the cycle. Obligations allow us to define this blocked-by relation uniformly for different blocking operations: a thread t is blocked by another thread t' if t is blocked on a blocking operation and t' holds an obligation to unblock it. For instance, t is blocked by t' if t tries to acquire a lock and t' holds the lock (and thus has an obligation to release it), or if t tries to join t' (and thus t' has an obligation to terminate).

We guarantee deadlock freedom by preventing cycles in the blocked-by relation. For this purpose, we introduce a strict partial order on threads and ensure via proof obligations for all blocking operations that a thread t may be blocked by a thread t' only if t is (strictly) less than t'. The order on threads is defined by letting the programmer define a strict partial *wait order* on obligations. A thread t is less than t' if for each obligation o held by t there exists an obligation o' held by t' such that o is less than o'. Cycles in the blocked-by relation are then prevented by proving for each blocking operation that each obligation held by the thread executing the blocking operation is less than the obligation to unblock it. Since this proof obligation refers only to the current thread, it can be checked in thread-modularly (we will discuss later how to check it procedure-modularly).

The wait order on obligations generalizes our earlier work [15] to arbitrary obligations. Like that work, we assume that the wait order on obligations is fixed throughout the execution of a program (but the order on threads changes dynamically as they obtain and lose obligations).

3 Programming and Assertion Language

In this section, we introduce the programming and assertion language. Their semantics will be defined in the next two sections.

$$
\begin{array}{rcl}
S & ::= & v := \texttt{new lock} \\
& | & \texttt{acquire}\ e \\
& | & \texttt{release}\ e \\
& | & v := \texttt{new}\ C \\
& | & \texttt{send}\ e_1(e_2) \\
& | & \texttt{receive}\ v := e \\
& | & \texttt{call}\ v := e_1.M(e_2) \\
& | & \texttt{fork}\ v := e_1.M(e_2) \\
& | & \texttt{join}\ v := e \\
& | & \texttt{while}(e)\ \texttt{invariant}\ A\ \{\ S_1\ \}
\end{array}
$$

Figure 4 The relevant statements of our programming language. We omitted assignment, sequential composition, and conditional statements because their treatment is straightforward. A fork statement yields a token, which can be used to join the forked thread.

3.1 Programming Language

We present our technique for a simple imperative programming language with iteration and recursion, threads, as well as dynamically-created locks and channels. For simplicity, we omit other heap-allocated objects because their treatment is orthogonal to the focus of this paper. However, our technique is compatible with permission-based program logics that handle them.

A program consists of a sequence of method declarations and channel type declarations. A method declaration has the form

```
method M(p: T1) returns (r: T2)
  requires A1;
  ensures A2;
{ S }
```

where M is a unique method name and each T_i is one of the following types: **bool**, **int**, **lock**, **token**, or a channel type. A_i are assertions and S is a statement, see below. Like in the Chalice language [15], a channel type declaration has the form

```
channel C(p: T) where A;
```

where C is a unique channel type name. Messages sent over such a channel are values of type T. The **where** clause specifies a *channel invariant*, that is, constraints on the messages; it also specifies the credits sent with each message.

Statements (Fig. 4) include operations on non-reentrant locks (creation, acquire, release), operations on channels (creation, send, receive), method call, thread fork and join, and loops with loop invariants. We also assume to have assignments, sequential composition, and conditional statements, but do not formalize them because they are straightforward. Expressions e include constants, variables v, and the usual boolean and arithmetic operations. We will explain and formalize the semantics of statements in Sec. 5.

For simplicity, channels have unbounded buffers such that send operations never block. Therefore, the blocking operations in our language are acquiring a lock, receiving a message, and joining a thread.

$$
\begin{array}{lcl}
A & ::= & e \\
  & | & A_1 \ \texttt{\&\&} \ A_2 \\
  & | & e \Rightarrow A_1 \\
  & | & \texttt{releases}(e_1, e_2) \\
  & | & \texttt{sends}(e_1, e_2, e_3) \\
  & | & \texttt{terminates}(e) \\
  & | & \texttt{joinable}(e) \\
  & | & \texttt{waitlevel} \ll e
\end{array}
$$

Figure 5 The assertion language. The three kinds of obligations exhibit all different characteristics of obligations discussed in Sec. 2.1.

3.2 Assertion Language

Assertions are used as method pre and postconditions, loop invariants, and channel invariants. Besides the usual constraints on variables, they specify which obligations and credits get transferred between method executions and loop iterations, along with their measures.

Measures. In order to define measures for obligations, we adopt Dafny's approach [13] and assume a pre-defined well-founded strict partial order $\sqsubset$ on all values of a program execution. For instance, for integers x and y, we define $x \sqsubset y \Leftrightarrow x < y \land 0 \leq y$, whereas for an integer x and a lock l, $x \sqsubset l$ is undefined. The resulting well-founded set forms a complete lattice $(\mathbb{V}, \sqsubset)$ with top element $\top$ and bottom element $\bot$. Assuming a pre-defined order simplifies the presentation of the verification technique. An adaptation to user-defined orders is possible, but reveals nothing interesting.

Wait levels. As explained in Sec. 2.3, we use a strict partial order on obligations to prove deadlock freedom. To encode this order, we assign every obligation a *wait level*, that is, a value in a dense lattice $(\mathbb{L}, \ll)$ with strict order $\ll$ and bottom element $\bot$.

Assertions. The assertion language is summarized in Fig. 5. It includes boolean expressions, conjunction, and implication. Moreover, there are assertions for three kinds of obligations. For a releases-obligation $\texttt{releases}(e_1, e_2)$, e_1 of type `lock` denotes the lock that must be released and $e_2 \in \mathbb{V}$ is the measure. For a sends-obligation $\texttt{sends}(e_1, e_2, e_3)$, e_1 is of a channel type and denotes the channel on which messages must be sent, e_2 is an integer that denotes how many messages must be sent, and e_3 is the measure. When e_2 is negative, the assertion denotes credits, that is, permissions to receive rather than obligations to send. For a terminates-obligation $\texttt{terminates}(e)$, e is the measure. For all three obligation assertions, the measure can be any value in $\mathbb{V}$, including $\top$ and $\bot$. The assertion $\texttt{joinable}(e)$, where e is of type `token` provides the permission to join the thread denoted by e. A thread may have a join-permission for e if the thread represented by the token e is guaranteed to terminate and has not been joined yet, and if no other thread has the permission to join it. The assertion $\texttt{waitlevel} \ll e$ expresses that the wait level of each obligation held by the current thread is strictly less than the wait level of e. Thus, the current thread may execute a blocking operation, where the corresponding obligation to unblock has level e, without creating a deadlock. We say that e *is above the current wait level if* $\texttt{waitlevel} \ll e$. One can think of

`waitlevel` as the maximum wait level of all obligations held by the current thread; however, we will use it to specify and check only upper bounds one these levels.

Conjunction `&&` is analogous to separating conjunction in separation logic. In particular, `releases`(l, n) `&&` `releases`(l, n) expresses that the current thread must release lock l twice. Since this is not possible for non-reentrant locks, the conjunction is equivalent to false. The conjunction `sends`$(c, 1, n)$ `&&` `sends`$(c, 1, n)$ expresses that the current thread has *two* obligations to send a message on channel c; that is, it is equivalent to `sends`$(c, 2, n)$.

Note that the use of sends-obligations and credits is not new [15] (see App. A for an example). We include them here to demonstrate how our technique handles a range of obligations uniformly and to exhibit all different characteristics of obligations discussed in Sec. 2.1. Sends-obligations can be accumulated, have the dual concept of sends-credits, and can be transferred between threads, whereas releases-obligations and terminates-obligations cannot be accumulated, have no credits, and cannot be transferred. Therefore, our assertion language is representative for a wide range of obligations including for instance obligations to await an asynchronous task or perform I/O.

Well-formedness Conditions. We impose several well-formedness conditions on assertions.

1. Method postconditions must not contain terminates-obligations because these obligations are satisfied when the method terminates and, thus, not returned to the caller.
2. A method may be forked only if its precondition does not contain any releases-obligations. This condition reflects that a lock must be released by the thread that acquired it; neither the held lock nor the releases-obligation can be transferred to another thread.
3. A method may be forked only if its postcondition does not contain any obligations. This condition prevents leaking of obligations when a forked thread is never joined. For terminates and releases-obligations, this condition can be checked syntactically. If the postcondition contains an assertion `sends`(c, e, n), we verify that e evaluates to a non-positive number, that is, the assertion denotes a credit.
4. A channel invariant must not contain any obligations (but credits are allowed). This condition ensures that obligations cannot be leaked by sending them in a message that might never get received.
5. A channel invariant must not contain wait level constraints because these constraints cannot be interpreted consistently in the sending and receiving thread of a message.

4 Encoding of Assertions

In this section, we present an encoding of assertions into a guarded command language similar to Boogie [1]. For readability, we use dedicated operators and constant symbols for measures and wait levels rather than Boogie's uninterpreted functions, and bulk updates (**foreach** statements) instead of encoding them via Boogie's `havoc` and `assume` statements. In the following, we introduce the representation of program states, explain how we encode the transfer of obligations, and then formalize the meaning of assertions.

4.1 Encoding of States

The state of a method execution consists of the method's parameter and result variables, its local variables, as well as the obligations (and credits) held by this method execution. To treat the different kinds of obligations uniformly, we introduce a type

$$\mathbf{obl} = \mathbf{lock} \cup \mathbf{channel} \cup \{\mathbf{term}\}$$

where **channel** includes all channel types declared in the program. Here, a lock identifies a releases-obligation, a channel identifies a sends-obligation (or credit), and the identifier **term** identifies a terminates-obligation. Using the **obl** type, we declare a global map that stores the obligations and credits held by the current method execution or loop iteration:

$$\mathcal{B} : \mathbf{obl} \to \mathbb{Z}$$

$\mathcal{B}[o] = n$ encodes that the current method execution has n obligations for o if n is positive, and $-n$ credits if n is negative, The latter occurs only if o is a channel.

As we explained in Sec. 2.2, we track separately which obligations are fresh, that is, have been obtained since the prestate of the current method execution or loop iteration. The number of fresh obligations is stored in a global map:

$$\mathcal{F} : \mathbf{obl} \to \mathbb{N}$$

$\mathcal{F}[o]$ yields how many of the obligations in $\mathcal{B}[o]$ are fresh. If there are no obligations, $\mathcal{F}[o]$ is zero. That is, the following invariants hold in all states:

$$\forall o \in \mathbf{obl} \cdot 0 \leq \mathcal{B}[o] \Rightarrow \mathcal{F}[o] \leq \mathcal{B}[o]$$
$$\forall o \in \mathbf{obl} \cdot \mathcal{B}[o] \leq 0 \Rightarrow \mathcal{F}[o] = 0$$

Since the first execution of a join statement for any thread t may block, we need in principle a credit that provides the permission to join t (see the characteristics of obligations in Sec. 2.1). This credit is the dual of the terminates-obligation for t. That is, the forker of t obtains the credit needed to join t if t promises to terminate, that is, consumes a terminates-obligation. It is possible to encode join-permissions as terminates-credits, but such an encoding complicates `terminates` assertions (which would need an argument that identifies the thread) and the encoding of fork (since terminates-obligations in the precondition of the forked method must be interpreted differently in the forker and in the forkee). Therefore, we choose a different encoding here. The map $\mathcal{B}$ does not contain termination information about threads other than the current thread; such information is stored in a separate map that yields whether a thread may be joined:

$$\mathcal{J} : \mathbf{token} \to \mathbb{B}$$

Finally, we record the wait level of each obligation in the following map, where $\mathbb{L}$ is the set of wait levels:

$$\mathcal{L} : \mathbf{obl} \cup \mathbf{token} \to \mathbb{L}$$

For a lock or channel o, $\mathcal{L}[o]$ denotes the wait level of the corresponding releases- or sends-obligations. $\mathcal{L}[\mathbf{term}]$ denotes the level of the terminates-obligation of the current thread, and for a token t, $\mathcal{L}[t]$ denotes the level of the terminates-obligation of the thread represented by t.

4.2 Transfer of Obligations and Credits

Our assertions do not only express conditions on the state but also specify which obligations (and credits) get transferred between method executions and loop iterations. This behavior is similar to assertions in permission logics, which describe how ownership of resources is transferred. We formalize the meaning of assertions via two operations, exhale and inhale (sometimes called produce and consume). In this subsection, we explain how to exhale and inhale obligations and credits. A key virtue of our approach is that these operations are uniform for all kinds of obligations. Exhaling and inhaling assertions will be explained in the next subsection.

$$
\begin{array}{l}
\mathit{Exhale}_{obl}(o, n, m, \mathit{creditsAllowed}, P) = \\
\quad \textbf{assert } \mathit{creditsAllowed} \lor n \leq \mathcal{B}[o]; \\
\quad \textbf{if } (m = \top) \; \{ \\
\quad\quad \textbf{assert } n \leq \mathcal{F}[o] \lor \mathcal{B}[o] \leq \mathcal{F}[o]; \\
\quad\quad \mathcal{F}[o] := \mathit{max}(\mathcal{F}[o] - n, 0); \\
\quad \} \textbf{ else } \{ \\
\quad\quad \textbf{assert } 0 < n \land \mathcal{F}[o] < \mathcal{B}[o] \Rightarrow m \sqsubset P[o]; \\
\quad \} \\
\quad \mathcal{B}[o] := \mathcal{B}[o] - n; \\
\quad \textbf{if } (\mathcal{B}[o] < \mathcal{F}[o]) \; \{ \\
\quad\quad \mathcal{F}[o] := \mathit{max}(\mathcal{B}[o], 0); \\
\quad \}
\end{array}
$$

Figure 6 The exhale operation for obligations and credits. $o \in \textbf{obl}$ is the obligation, $n \in \mathbb{Z}$ indicates the number of obligations (or credits) to exhale, and $m \in \mathbb{V}$ is the measure of the obligations to be exhaled. The boolean flag *creditsAllowed* indicates whether credits are allowed for the kind of obligations to be exhaled. $P \in \textbf{obl} \to \mathbb{V}$ provides the measure of obligations in the prestate of the enclosing method or loop for the check that the measure decreases.

Exhale. Exhaling obligations is formalized in Fig. 6. $\mathit{Exhale}_{obl}(o, n, m, \mathit{creditsAllowed}, P)$ exhales n obligations (or $-n$ credits, if n is negative) for o (where o is a lock, channel, or **term**) with measure m. It first asserts that credits are permitted for this kind of obligation or that the current state has enough obligations to exhale. (Applications of Exhale_{obl} will ensure that *creditsAllowed* is true if n is negative.)

For the rest of the operation, let us first consider the case that we exhale obligations, that is, $0 < n$. If the exhaled obligations are fresh (indicated by $m = \top$), we check that there are enough fresh obligations available or that there are no non-fresh obligations. In the former case, the fresh obligations are given away. In the latter case, the exhale gives away all available fresh obligations and obtains some credits. It would be unsound to exhale fresh obligations if neither case applied because reducing the number of obligations would then treat non-fresh obligations as fresh, thereby providing a way to postpone their satisfaction. If the exhaled obligations are non-fresh ($m \neq \top$), we check that if the current state holds non-fresh obligations ($\mathcal{F}[o] < \mathcal{B}[o]$), their measure decreased w.r.t. the prestate measure of the enclosing method or loop, provided by the map $P \in \textbf{obl} \to \mathbb{V}$. In both cases, we remove the exhaled obligations from the state and adjust the number of fresh obligations to maintain the invariants mentioned in Sec. 4.1.

If we exhale credits (that is, $n \leq 0$), the assertions in both branches of the conditional hold trivially (recall that $0 \leq \mathcal{F}[o]$). Giving away fresh credits increases the number of fresh obligations, and giving away any credits always increases the number of obligations. It is therefore preferable to make all credits in assertions fresh.

Creation and Cancellation of Credits. A sends-credit for a channel c is created by exhaling a sends-obligation for c in a state that holds no such obligation. However, the inverse operation—canceling an obligation with a credit— is *not* permitted. That is, inhaling a credit in a state that holds a corresponding obligation, or inhaling an obligation in a state that holds a corresponding credit leads to a verification error. It would be unsound to create a credit by exhaling an obligation with a small measure and then cancel the credit with

$$
\begin{aligned}
& \mathit{Inhale}_{obl}(o, n, m, P) = \\
& \quad \textbf{if } (0 < n)\ \{ \\
& \quad\quad P[o] := P[o] \sqcap m;\ \} \\
& \quad \} \\
& \quad \textbf{assert } (0 < n \Rightarrow 0 \leq \mathcal{B}[o]) \wedge (n < 0 \Rightarrow \mathcal{B}[o] \leq 0); \\
& \quad \mathit{Exhale}_{obl}(o, -n, m, \texttt{true}, P_\top);
\end{aligned}
$$

Figure 7 The inhale operation for obligations and credits. The parameters o, n, and m are analogous to Exhale_{obl}. Inhaling obligations records their measures in the map $P \in \textbf{obl} \to \mathbb{V}$ for later checks. Note that we treat the inhale operation as a parameterized macro such that updates to P modify the argument map at the call site. $P_\top \in \textbf{obl} \to \mathbb{V}$ yields $\top$ for all obligations and is used to suppress the measure check in Exhale_{obl}, which is not needed during inhale.

an obligation that has a larger measure. This would effectively increase the measure of the obligation and, thus, provide a way to postpone the satisfaction of the obligation indefinitely. Even if the obligations involved in creating and canceling a credit had the same measure, one could postpone the satisfaction of the obligation indefinitely by arranging a sequence of threads where each thread obtains a credit from its successor to cancel its own obligation, creating another obligation in the successor, and so on.

One could prevent this unsoundness by recording the measure of the exhaled obligation when creating a credit and then enforcing that the credit may cancel only obligations that have a strictly larger measure. Since this solution requires substantial bookkeeping and since the main purpose of sends-credits is to enable receive operations (rather than canceling sends-obligations), we simply forbid cancellation of obligations and credits altogether. This rule is reflected in the encoding of inhale below.

Inhale. Inhaling obligations is formalized in Fig. 7. $\mathit{Inhale}_{obl}(o, n, m, P)$ inhales n obligations (or $-n$ credits, if n is negative) for o with measure m. The operation records the measures of inhaled obligations in map P. If there are multiple obligations for o, we abstract their measures by storing their minimum. This is achieved by using the meet $\sqcap$ of the measure lattice. We treat the inhale operation as a parameterized macro such that updates to P modify the argument map at the call site (that is, P behaves like an in-out parameter). We will record measures only in the prestates of method executions and loop iterations; in all other cases, we will pass a dummy map for P.

The assertion after the update of P prevents credit cancellation as explained above. Finally, obligations are added by exhaling the corresponding credits, and vice versa. Since the decrease-checks for measures before recursive calls and at the end of loop iterations will be encoded via exhale, inhale does not have to perform any such checks. Therefore, it passes $P_\top$, which yields $\top$ for all obligations, to Exhale_{obl}, such that the check $m \sqsubset P[o]$ there will trivially succeed.

4.3 Exhaling and Inhaling Assertions

Exhaling an assertion A checks that the constraints specified by A hold and removes the obligations and credits specified in A from the current state. The definition is provided in Fig. 8. Exhaling proceeds in two phases. The first phase checks all constraints except those on wait level and handles the transfer of obligations and credits. The second phase only checks

$$
\begin{array}{lcl}
\mathit{Exhale}(A, P) & = & \mathit{Exhale}_1(A, P)\ \mathit{Exhale}_2(A, _) \\
\\
\mathit{Exhale}_i(A_1\ \texttt{\&\&}\ A_2, P) & = & \mathit{Exhale}_i(A_1, P)\ \mathit{Exhale}_i(A_2, P) \\
\mathit{Exhale}_i(e \Rightarrow A, P) & = & \textbf{if}\ (\llbracket e \rrbracket)\ \{\ \mathit{Exhale}_i(A, P)\ \} \\
\\
\mathit{Exhale}_1(e, _) & = & \textbf{assert}\ \llbracket e \rrbracket; \\
\mathit{Exhale}_1(\texttt{releases}(e_1, e_2), P) & = & \mathit{Exhale}_{obl}(\llbracket e_1 \rrbracket, 1, \llbracket e_2 \rrbracket, \texttt{false}, P) \\
\mathit{Exhale}_1(\texttt{sends}(e_1, e_2, e_3), P) & = & \mathit{Exhale}_{obl}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket, \llbracket e_3 \rrbracket, \texttt{true}, P) \\
\mathit{Exhale}_1(\texttt{terminates}(e), P) & = & \mathit{Exhale}_{obl}(\textbf{term}, 1, \llbracket e \rrbracket, \texttt{true}, P) \\
\mathit{Exhale}_1(\texttt{joinable}(e), _) & = & \textbf{assert}\ \mathcal{J}[\llbracket e \rrbracket]; \mathcal{J}[\llbracket e \rrbracket] := \texttt{false}; \\
\\
\mathit{Exhale}_2(\texttt{waitlevel} \ll e, _) & = & \textbf{assert}\ \mathit{levelBelow}(\mathcal{B}, \mathcal{L}[\llbracket e \rrbracket]);
\end{array}
$$

Figure 8 Encoding of exhale. A is an assertion, and $P \in \textbf{obl} \to \mathbb{V}$ provides the prestate measures for the check that obligation measures decrease. All cases not mentioned here are defined as **skip**. $\llbracket _ \rrbracket$ encodes expressions of the programming language; it is straightforward and, therefore, omitted.

wait level constraints. This encoding via two phases is necessary to treat wait level constraints soundly. It checks wait level constraints during exhale *after* obligations and credits have been removed from the state, and assumes wait level constraints during inhale *before* obligations and credits have been added (see Fig. 9 below). That is, in both cases, `waitlevel` refers to a state that does not contain the transferred obligations and credits. This fixes an unsoundness in Chalice [15], where it was possible to interpret `waitlevel` inconsistently during exhale and inhale and, thus, exhale assertions that lead to an inconsistency when inhaled.

In both phases of exhale, conjunction is treated multiplicatively by sequentially exhaling the two conjuncts. This is analogous to an encoding of separating conjunction [21]. Implication is encoded via a conditional statement.

Phase 1 uses Exhale_{obl} from Fig. 6 to transfer obligations and credits, and to check that measures decrease. Even though there are no terminates-credits, we set the *creditsAllowed* parameter of Exhale_{obl} to true for terminates-obligations because our encoding of statements will lead to intermediate states with a negative number of terminates-obligations. Exhaling a join-permission asserts that such a permission is held and removes it.

Phase 2 checks wait level constraints. In order to be useful to prove deadlock freedom, `waitlevel` $\ll e$ expresses that the wait level of each obligation *held by the current thread* is strictly less than the wait level of e. In our procedure-modular verification technique, we cannot check this condition directly because we record (in map $\mathcal{B}$) only the obligations *held by the current method execution or loop iteration*, but not those held by other method executions on the call stack or enclosing loops. To account for those, our encoding uses a local variable $\mathit{residue} \in \textbf{token}$ in each method. We leave the value of *residue* unspecified, but ensure that we can prove that its level is less than an upper bound u only if the level of all obligations held by the current thread, but not by the current method execution or loop iteration, is less than u. Therefore, we can prove `waitlevel` $\ll e$ by proving that the level of all obligations recorded in $\mathcal{B}$ as well as the level of *residue* are less than e's level. We encode this via the following predicate:

$$\mathit{levelBelow}(B, u) = (\forall o \in \textbf{obl} \cdot 0 < B[o] \ \Rightarrow\ \mathcal{L}[o] \ll u) \ \wedge\ \mathcal{L}[\mathit{residue}] \ll u$$

Our encoding ensures that the only information obtained about *residue*'s level is the upper

$$
\begin{array}{lcl}
\mathit{Inhale}(A, P) & = & \texttt{var } \mathcal{B}_{old} := \mathcal{B};\, \mathit{Inhale}_1(A, P) \\
\\
\mathit{Inhale}_1(A_1 \texttt{ \&\& } A_2, P) & = & \mathit{Inhale}_1(A_1, P)\ \mathit{Inhale}_1(A_2, P) \\
\mathit{Inhale}_1(e \Rightarrow A, P) & = & \textbf{if } (\llbracket e \rrbracket)\ \{\ \mathit{Inhale}_1(A, P)\ \} \\
\mathit{Inhale}_1(e, _) & = & \textbf{assume } \llbracket e \rrbracket; \\
\mathit{Inhale}_1(\texttt{releases}(e_1, e_2), P) & = & \mathit{Inhale}_{obl}(\llbracket e_1 \rrbracket, 1, \llbracket e_2 \rrbracket, P) \\
\mathit{Inhale}_1(\texttt{sends}(e_1, e_2, e_3), P) & = & \mathit{Inhale}_{obl}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket, \llbracket e_3 \rrbracket, P) \\
\mathit{Inhale}_1(\texttt{terminates}(e), P) & = & \mathit{Inhale}_{obl}(\textbf{term}, 1, \llbracket e \rrbracket, P) \\
\mathit{Inhale}_1(\texttt{joinable}(e), _) & = & \mathcal{J}[\llbracket e \rrbracket] := \texttt{true}; \\
\mathit{Inhale}_1(\texttt{waitlevel} \ll e, _) & = & \textbf{assume } \mathit{levelBelow}(\mathcal{B}_{old}, \mathcal{L}[\llbracket e \rrbracket]);
\end{array}
$$

Figure 9 Encoding of inhale. A is an assertion, and $P \in \mathbf{obl} \to \mathbb{V}$ is used to record the measures of inhaled obligations.

bounds when inhaling wait level constraints as part of method pre or postconditions, or loop invariants. Therefore, the prover needs check assertions for any value of *residue*'s level below these upper bounds, including a value above the levels of the obligations held by the transitive callers of the current method. To understand why such a value always exists, consider a method m with precondition `waitlevel` $\ll e$. This condition constrains the level of m's *residue* variable to be less than e's level. When exhaling this precondition in the caller n, we check that the levels of all obligations held by n are less than e's level. Therefore, since wait levels form a dense lattice, there exists a possible value for the level of m's *residue* variable that is above all obligations held by n and less than e's level. By checking (as part of exhaling the precondition) that the level of n's *residue* variable is less than e's level, we know that there exists a value for the level of m's *residue* variable that is above the level of n's *residue* variable and less than e's level. The argument applies inductively to n's *residue* variable, the one in n's caller, and so on. That is, m's *residue* also reflects the obligations held by those method executions. The argument is analogous for enclosing loops.

The definition of inhale in Fig. 9 is analogous to exhale. It stores the current obligations map $\mathcal{B}$ before transferring obligations or credits in order to interpret wait level constraints consistently with exhale. Inhaling a constraint assumes it. Obligations and credits are transferred using the Inhale_{obl} macro from Fig. 7, and join-permissions are inhaled by adding them.

5 Encoding of Methods and Statements

In this section, we present the proof rules for our verification technique via an encoding into Boogie [1]. The resulting Boogie program contains neither obligations (which are encoded by accesses to the maps $\mathcal{B}$ and $\mathcal{F}$) nor exhale and inhale operations (which are replaced by their definitions). Therefore, we can verify the program by computing weakest preconditions over the guarded commands and proving them in an SMT solver. Verification is procedure and thread-modular. That is, each method is verified without considering its caller or interference from other threads.

$\langle\!\langle$`method` $M(p)$ `returns` (r) `requires` $pre_M(\mathbf{this}, p)$ `ensures` $post_M(\mathbf{this}, p, r)$ $\{\ S\ \}\rangle\!\rangle =$

$\quad$ **assume** $\forall o \in \mathbf{obl} \cdot \mathcal{B}[o] = 0$;
$\quad$ **var** *residue*;
$\quad$ **var** $P_{method} := P_\top$;
$\quad$ $Inhale(pre_M(\mathbf{this}, p), P_{method})$
$\quad$ **foreach** $o \in \mathbf{obl}$ $\{\ \mathcal{F}[o] := 0;\ \}$
$\quad$ $[\![S]\!]$
$\quad$ $Exhale(post_M(\mathbf{this}, p, r), P_\top)$
$\quad$ $\mathcal{B}[\mathbf{term}] := 0$;
$\quad$ **assert** $\forall o \in \mathbf{obl} \cdot \mathcal{B}[o] \leq 0$;

Figure 10 Encoding $\langle\!\langle_\rangle\!\rangle$ of methods. The assertions $pre_M(\mathbf{this}, p)$ and $post_M(\mathbf{this}, p, r)$ are the method precondition and postcondition, resp.

5.1 Methods

Fig. 10 shows the encoding of methods. Before inhaling the precondition, the execution of a method holds neither obligations nor credits. The value of the local variable *residue* is unspecified; its level is constrained when inhaling wait level constraints. The subsequent inhale operation assumes the method precondition and transfers obligations and credits from the caller to the callee. It records the measures of obligations in a map P_{method}, which will be used in call and fork statements to ensure that measures decrease. The recording works by passing the all-top map $P_\top$ into the inhale macro, which, for each inhaled obligation, takes the minimum (that is, the meet) of the stored measure and the measure of the inhaled obligation (see Fig. 7). After inhaling the precondition, we make all fresh obligations non-fresh since obligations that are fresh to the caller are not fresh to the callee as they existed before the execution of the callee started. This step is necessary to prevent fresh obligations from being transferred indefinitely from method execution to method execution.

The method body is encoded using the encoding function for statements $[\![_]\!]$. After executing the body, we exhale the postcondition. During this exhale, we do not need to check that measures have decreased (which happens only at call and fork sites and at the end of loop iterations). Therefore, we pass the all-top map $P_\top$ as last argument to the exhale operation such that the decrease-check succeeds trivially. After the exhale, we remove the terminates-obligation from the obligation map since the method is about to terminate. The final step is the leak check: upon termination, the method may hold no obligations. That is, all obligations passed in from the caller or obtained during the execution of the method must be satisfied, transferred to other threads (during a fork), or returned to the caller when exhaling the postcondition.

5.2 Call, Fork, and Join

A call statement (Fig. 11) is verified by exhaling the precondition of the callee and then inhaling its postcondition. The exhale needs to check that the measures of exhaled obligations decreased since the prestate of the caller. This is achieved by passing the measures from this state (variable P_{method}, which is initialized at the beginning of the enclosing method, see Fig. 10) into the exhale operation. After the exhale, we assert that the caller retains no obligations unless the callee promises to terminate. This assertion ensures obligations cannot be left behind in the caller in cases where the control flow might never return. The

$$
\begin{array}{lcl}
[\![\texttt{call}\ v := e_1.M(e_2)]\!] & = & \textbf{var}\ \mathit{term} := \mathcal{B}[\textbf{term}]; \\
 & & \mathit{Exhale}(\mathit{pre}_M([\![e_1]\!], [\![e_2]\!]), P_{\mathit{method}}) \\
 & & \textbf{assert}\ \forall o \in \textbf{obl} \cdot \mathcal{B}[o] \leq 0 \vee \mathcal{B}[\textbf{term}] < \mathit{term}; \\
 & & \mathcal{B}[\textbf{term}] := \mathit{term}; \\
 & & \mathit{Inhale}(\mathit{post}_M([\![e_1]\!], [\![e_2]\!], v), P_d) \\
 & & \\
[\![\texttt{fork}\ v := e_1.M(e_2)]\!] & = & \textbf{var}\ \mathit{term} := \mathcal{B}[\textbf{term}]; \\
 & & \mathit{Exhale}(\mathit{pre}_M([\![e_1]\!], [\![e_2]\!]), P_{\mathit{method}}) \\
 & & \textbf{havoc}\ v; \textbf{assume}\ \mathcal{L}[v] = \bot; \\
 & & \mathcal{J}[v] := (\mathcal{B}[\textbf{term}] < \mathit{term}); \\
 & & \mathcal{B}[\textbf{term}] := \mathit{term}; \\
 & & \textbf{havoc}\ w; \textbf{assume}\ \mathit{levelBelow}(\mathcal{B}, w); \\
 & & \mathcal{L}[v] := w; \\
 & & \\
[\![\texttt{join}\ v := e]\!] & = & \textbf{assert}\ \mathit{levelBelow}(\mathcal{B}, [\![e]\!]); \\
 & & \textbf{assert}\ \mathcal{J}[[\![e]\!]]; \\
 & & \mathit{Inhale}(\mathit{post}_e(v), P_d) \\
 & & \mathcal{J}[[\![e]\!]] := \texttt{false};
\end{array}
$$

Figure 11 Encoding of call, fork, and join statements. $P_d \in \textbf{obl} \to \mathbb{V}$ is a dummy map that is never read. The function post_e yields the postcondition of the method that was forked to obtain token e. We assume that the receiver and arguments of the fork are stored in the token, but omit this aspect in the encoding.

condition $\mathcal{B}[\textbf{term}] < \mathit{term}$ expresses that the callee promises to terminate. In this case, exhaling its precondition will transfer a terminates-obligation from the caller to the callee, that is, decrease the value of $\mathcal{B}[\textbf{term}]$ compared to the value before the exhale (stored in local variable *term*). Finally, since the assertion after the exhale quantifies over all obligations, including terminates-obligations, it enforces that the callee promises to terminate if the caller does (otherwise the caller would still hold its terminates-obligation after the exhale). Since terminates-obligations must be satisfied by each individual method and cannot be delegated, we restore the terminates-obligations after the exhale. The final inhale does not have to record measures since this is necessary only in the prestate of a method execution or loop iteration; therefore, it uses the dummy map P_d, which is never read from.

Allowing the caller to retain obligations when calling a terminating method is crucial for modularity; otherwise, the callee's precondition would have to mention different obligations for different call sites. Nevertheless, these obligations are accounted for in variable *residue* and, thus, affect wait level constraints. In particular, it is not possible for a caller to hold an obligation to unblock its callee (which might create a deadlock) because the obligation in the caller affects the wait level of the callee (via *residue*) and, thus, prevents the callee from executing the blocking operation (see for instance the first assertion in the encoding of join statements in Fig. 11).

The encoding of a fork statement is similar to a call. In particular, the measures of transferred obligation must decrease to ensure that they cannot be transferred from thread to thread indefinitely. However, since the forked method will be executed in a new thread, there are no restrictions on the obligations that remain in the forker. After the exhale, we pick a fresh token for the new thread. The fact that this token is different from existing

$$\begin{array}{lcl}
[\![v := \texttt{new}\ \textbf{lock}]\!] & = & \textbf{havoc}\ v; \textbf{assume}\ \mathcal{L}[v] = \bot; \\
 & & \textbf{havoc}\ w; \textbf{assume}\ \mathit{levelBelow}(\mathcal{B}, w); \\
 & & \mathcal{L}[v] := w; \\
 & & \mathcal{B}[v] := 0; \mathcal{F}[v] := 0; \\
[\![\texttt{acquire}\ e]\!] & = & \textbf{assert}\ \mathit{levelBelow}(\mathcal{B}, [\![e]\!]); \\
 & & \mathit{Inhale}_{obl}([\![e]\!], 1, \top, P_d) \\
[\![\texttt{release}\ e]\!] & = & \mathit{Exhale}_{obl}([\![e]\!], 1, \bot, P_\top)
\end{array}$$

Figure 12 Encoding of lock operations.

token is encoded by assuming that its level in the wait order is $\bot$, whereas all tokens for existing threads are implicitly assumed to have larger levels. The new thread can be joined if it promises to terminate, that is, if the forker's terminates-obligations get decreased by exhaling the forked method's precondition. Like for calls, the terminates-obligations get restored afterwards. Finally, we choose a wait level for the new thread that is above the current wait level, which will allow the current thread to join it later.

Since join is a blocking operation, it asserts that the token of the thread to be joined is above the current wait level (to avoid deadlock) and that the current thread has the appropriate join-permission (to avoid waiting on a non-terminating thread). We then inhale the joined method's postcondition and remove the join-permission to prevent a thread from being joined more than once, which could forge credits in the postcondition.

5.3 Lock Operations

The encoding of lock operations is presented in Fig. 12. To focus on the essentials, we do not associate locks with an invariant. An extension is straightforward, but requires that the invariant does not contain obligations (credits are permitted) [15]. Otherwise, a thread could get rid of its obligations by storing them in a lock, which might never get acquired again.

Creating a new lock picks a fresh **lock** value. The fact that this value is different from existing locks is encoded by assuming that its level in the wait order is $\bot$, whereas all other locks are assumed to have larger levels. The new lock is then inserted into the wait order above the current wait level, which allows the current thread to acquire it (specifying different levels for the new lock is possible [15], but omitted here for simplicity). Initially, the current thread does not hold any obligations for the new lock.

Acquiring a lock checks that the lock is above the current wait level to prevent deadlock. It then inhales a fresh releases-obligation for the lock to ensure that the acquired lock will eventually be released. Inhaling this obligation implicitly raises the current thread's wait level.

Releasing a lock exhales the corresponding releases-obligation. This exhale operation does not have to check that the obligation measure has been decreased. We achieve that by passing a non-$\top$ measure for the obligation (here, $\bot$) and $P_\top$ for the prestate map, such that the decrease-check succeeds trivially (since $\bot \sqsubset \top$).

$$
\begin{array}{lcl}
[\![v := \texttt{new}\ C]\!] & = & \textbf{havoc}\ v; \textbf{assume}\ \mathcal{L}[v] = \bot; \\
& & \textbf{havoc}\ w; \textbf{assume}\ levelBelow(\mathcal{B}, w); \\
& & \mathcal{L}[v] := w; \\
& & \mathcal{B}[v] := 0; \mathcal{F}[o] := 0 \\
\\
[\![\texttt{receive}\ v := e]\!] & = & \textbf{assert}\ levelBelow(\mathcal{B}, [\![e]\!]); \\
& & \textbf{assert}\ \mathcal{B}[[\![e]\!]] < 0; \\
& & Exhale_{obl}([\![e]\!], -1, \bot, _) \\
& & Inhale(inv([\![e]\!], v), P_d) \\
\\
[\![\texttt{send}\ e_1(e_2)]\!] & = & Exhale_{obl}([\![e_1]\!], 1, \bot, P_\top) \\
& & Exhale(inv([\![e_1]\!], [\![e_2]\!]), P_\top)
\end{array}
$$

Figure 13 Encoding of channel operations. C is a channel type, and *inv* denotes its invariant, which may refer to the channel itself, for instance, to denote sends-credits for the channel.

5.4 Message Passing

The encoding of channel operations is presented in Fig. 13. Channel creation is analogous to lock creation (see Fig. 12). Since receive is a blocking operation, we first assert that the channel is above the current wait level. Moreover, to ensure that some thread has an obligation to send on the channel (or has sent already), we require that the current thread has a sends-credit, which is subsequently consumed by exhaling it. Finally, we inhale the channel invariant *inv*, without recording any obligations measures. Since we assume sending to be a non-blocking operation, we simply exhale a sends-obligation (which got satisfied) and exhale the channel invariant.

Our well-formedness conditions (Sec. 3.2) ensure that channel invariants do not contain obligations. Therefore, it is neither possible to get rid of obligations by sending them in a message that is never received, nor to send obligations in circles indefinitely. It is also not possible to transfer obligations indirectly from one thread to another by sending a credit in the opposite direction. Such an indirect transfer would have to cancel the obligation in the receiver of the message with the credit contained in the message, which is prevented by our definition of inhale (see Sec. 4.2).

5.5 Loops

The encoding of loops (Fig. 14) includes both the representation of the loop within the enclosing code and the verification of the loop body. The two aspects are encoded by a non-deterministic choice (`if(*)`). The former resembles the encoding of a method call, whereas the latter is similar to a method body.

In both cases, we proceed by exhaling the loop invariant. This exhale does not check obligation measures since the measures of the loop encoded here are independent of the measures used by the enclosing method or loop (if any). Hence, we pass the all-top map $P_\top$ to the exhale operation. The check after exhaling the loop invariant ensures that the code before the loop does not retain any obligations unless the loop promises to terminate. This assertion is identical to the one for calls (Fig. 11). In particular, it enforces that the loop must promise to terminate if the enclosing loop or method has a terminates-obligation. We

$[\![$`while`(e) `invariant` A $\{$ S $\}]\!] =$
    var $term := \mathcal{B}[\textbf{term}]$;
    $\mathit{Exhale}(A, P_\top)$
    assert $\forall o \in \textbf{obl} \cdot \mathcal{B}[o] \leq 0 \vee \mathcal{B}[\textbf{term}] < term$;
    havoc *loop targets*;
    if $(*)$ $\{$
      $\mathit{Inhale}(A, P_d)$
      assume $\neg[\![e]\!]$;
    $\}$ **else** $\{$
      havoc $\mathcal{B}, \mathcal{F}, \mathit{residue}$;
      assume $\forall o \in \textbf{obl} \cdot \mathcal{B}[o] = 0$;
      var $P_{loop} := P_\top$;
      $\mathit{Inhale}(A, P_{loop})$
      foreach $o \in \textbf{obl}$ $\{$ $\mathcal{F}[o] := 0;$ $\}$
      assume $[\![e]\!]$;
      $[\![S]\!]$
      $\mathit{Exhale}(A, P_{loop})$
      assert $\forall o \in \textbf{obl} \cdot \mathcal{B}[o] \leq 0$;
      assume `false`;
    $\}$

Figure 14 Encoding of while statements. The first branch of the non-deterministic choice encodes the loop within the enclosing code and resembles a method call. The second branch verifies the loop body and resembles the encoding of a method.

then havoc the loop targets, that is, all local variables that get assigned to in the loop body. Any information about these variables that should be retained must be included in the loop invariant.

To represent the loop within the enclosing code, we simply inhale the loop invariant (without recording obligation measures), assume that the loop condition is false, and proceed to the statements after the loop.

To verify the loop body, we consider an arbitrary loop iteration. We first havoc the obligation maps and *residue* to remove any information from before the loop. The following steps are analogous to the encoding of methods (Fig. 10): Before inhaling the loop invariant, the loop iteration holds neither obligations nor credits. Then we inhale the loop invariant and record obligation measures in a map P_{loop} for the decrease-check at the end of the loop body. Finally, we make all fresh obligations non-fresh (to prevent them from being transferred indefinitely from iteration to iteration), and execute the loop body. After the loop body, we exhale the loop invariant, checking that obligation measures decreased during the loop body, and perform the same leak check as for methods. Finally, we stop verification by assuming false, in order to prevent verification from proceeding with the code after the loop (which is done in the other branch of the non-deterministic choice).

6 Soundness

Our technique guarantees that in any execution of a verified program, no thread blocks indefinitely. This guarantee holds under the assumptions that (1) all thread transitions are

strongly fair and (2) the number of threads in each execution state is finite. A strongly-fair transition executes infinitely often if it is enabled infinitely often. Hence, we make the assumption that the thread scheduler ensures strong fairness and that we have fair locks and fair message reception. The number of threads in each state must be finite to prevent infinite chains of threads where each thread is blocked by its successor and, thus, never gets unblocked. This requirement is met by any execution platform with finite memory; to implement it, a fork operation aborts the entire program execution when a certain (unknown) number of threads is reached. Note that the number of threads is finite, but unbounded. That is, verification guarantees finite blocking for program executions with an arbitrary finite number of threads in each state. In this section, we provide the main arguments why our technique is sound.

The following properties hold in each execution state of a verified program:

1. *A thread t holds a lock l iff t has a releases-obligation for l.* This property is preserved by all lock operations (Fig. 12). The other operations preserve it because they neither add nor remove releases-obligations. In particular, our well-formedness conditions (Sec. 3.2) ensure that releases-obligations cannot be transferred to another thread during fork, join, or message passing.
2. *For each channel c, the total number of credits in the system (that is, held by a thread or stored in a message) is at most the total number of obligations plus the number of messages stored in c's buffer.* This inequality is preserved by all channel operations (Fig. 13). For all other operations, each exhale has a corresponding inhale, keeping the total number of obligations and credits in the system constant. The only exception is exhaling the postcondition of a forked method if the thread does not get joined. However, our well-formedness conditions ensure that postconditions of forked methods do not contain obligations. Moreover, our leak checks ensure that the termination of method executions and loop iterations maintains the number of obligations in the system and does not increase the number of credits, thus, preserving the inequality.
3. *If a thread t has a join-permission for a thread t' then t' has a terminates-obligation or has terminated already.* Fork and join (Fig. 11) preserve the property. In particular, fork provides a join-permission only if the new thread promises to terminate, and join removes this permission. Moreover, a thread keeps its terminates-obligation until the forked method terminates.
4. *If a thread t is blocked, the number of obligations to unblock it held by all other threads is positive.* This property follows from the encoding of the three blocking statements and Properties 1–3.
5. *There is no cycle among threads such that each thread on the cycle waits for the next one to unblock it; that is, there is no deadlock.* Each blocking statement checks that the wait level of the current thread is strictly less than the wait level of the thread that must unblock it, that is, the thread that (a) holds the lock to be acquired (since held locks contribute to the wait level by Property 1), (b) has a sends-obligation for the channel on which to receive, or (c) needs to terminate (since the thread's current wait level is no smaller than its initial wait level, which is the level of its token).

The following properties hold for each execution of a verified program:

6. *A fresh obligation gets satisfied or becomes non-fresh within finitely many execution steps.* A single thread t can hold on to a fresh obligations only for a finite number of steps because every fresh obligation becomes non-fresh at the beginning of each method or loop body, that is, before the thread can transfer the obligations to another thread.

7. *A non-fresh obligation gets satisfied within finitely many execution steps.* A non-fresh obligation cannot stay in one thread forever since its measure must decrease for each recursive call or loop iteration. It can be transferred to other threads only via fork, which also checks that the measure decreases. The well-formedness conditions ensure that transfers through join or message passing are not possible.

These properties imply soundness as follows. Whenever there is a blocked thread t_0 then there is a sequence $t_0, t_1, \ldots$ such that t_{i+1} has an obligation to unblock t_i. By the assumption that the number of threads is finite, this sequence is finite. By Properties 4 and 5, its last thread t_n is not blocked, that is, is enabled. By the assumption of fair scheduling, t_n will eventually make progress and, by Properties 6 and 7, its obligation will eventually be satisfied, unblocking thread t_{n-1}. Thread t_{n-1} might re-block immediately if another thread acquires the lock or receives the message t_{n-1} is waiting for. However, since we assume fair locks and message reception, enabling t_{n-1} infinitely often ensures that it will make progress eventually. Therefore, the argument applies inductively.

7 Related Work

Chalice. The work most closely related to ours is Leino et al.'s approach to verifying deadlock freedom in Chalice [15]. However, their verification technique uses a partial correctness semantics and, thus, provides no guarantees for the common case that a program contains non-terminating threads. It also does not support termination proofs. In contrast, the key contribution of our work is a technique to prove finite blocking even in the presence of non-terminating threads, and this technique subsumes termination checking. Leino et al. handle blocking receive statements via credits and obligations (called debt). We generalize this idea to arbitrary blocking operations, which gives us a uniform treatment of locks, channels, and thread join, and provides a systematic way to encode further blocking operations. This uniform treatment also allows us to replace several ad-hoc solutions in Chalice such as holds-predicates and lockchange-clauses [14]. We adopted the general approach of preventing deadlock via a wait order that includes locks, channels, and threads from Chalice. However, the encoding of wait level constraints presented by Leino et al. is unsound because it does not interpret `waitlevel` consistently during exhale and inhale. Our encoding fixes this problem via the 2-phase exhale and a consistent interpretation during inhale.

Liveness. Finite blocking and termination are liveness properties that can be proved using linear-time temporal logic [17]. For instance, Manna et al. [18] verify liveness properties of concurrent programs running an arbitrary number of (identical) threads. In contrast to this work, we present a methodology based on obligations that provides a strategy how to structure specifications and proofs. In particular, our technique supports modular verification, where each method is verified without knowledge of their callers or concurrently executing threads. Like our work, Manna et al. use strong fairness as one of their fairness notions.

Gotsman et al. [8] present a verification technique to show that a non-blocking algorithm is wait-free, lock-free, or obstruction-free. These liveness properties are checked by proving termination of an arbitrary number of operations running in parallel. The authors use a rely-guarantee logic to reason about the interference between these parallel executions, which is non-modular. Our work focuses on blocking operations. In this context, we can use specifications based on obligations and credits to make verification modular.

Model checkers are able to verify general temporal logic properties in LTL or CTL, including liveness properties. Many model checkers bound the number of threads (such as

SPIN [9]) or the number of context switches (such as CHESS [19]), whereas our technique verifies programs for any finite number of threads and any number of context switches. Software model checking can also be applied to infinite state programs by utilizing different (automatic) abstraction techniques [2]. In contrast to these approaches, our technique is procedure-modular, which makes it applicable to libraries and improves scalability, at the price of having to write specifications.

Termination. Our technique is closely related to existing work on termination checking. However, it goes beyond termination checking in two major ways. First, it allows one to prove finite blocking in concurrent programs, which includes termination checking as a special case. In particular, finite blocking requires a solution that distinguishes safe implementations where a thread unblocks another thread and then obtains yet another obligation to unblock (for instance, by releasing and re-acquiring a lock) from unsafe situations where a thread continues to block another thread. Such situations do not occur during termination checking. Second, our technique handles different kinds of obligations and supports the dual notion of credit. In particular, credits may be transferred between threads, which requires extra checks to prevent unsound cancellation. Again, this problem does not occur in termination checking.

Le et al. [12] propose a verification logic for termination and non-termination. Similar to our work, their logic uses a resource that reflects termination and that is manipulated similarly to permissions in permission logics. Le et al. associate their termination resources with upper and lower bounds on their lifetimes, which allows them to prove termination as well as definite non-termination.

We adopted Dafny's approach to obtain measures by defining a well-founded order on all values of a program execution [13]. Dafny lifts this order to define a lexicographic order on sequences of values and includes the import relation among modules as a part of this order. These extensions are compatible with our use of measures.

There exist powerful automated termination checkers for both sequential and concurrent programs [2, 3, 4, 6]. The focus of most work in this area is on inferring termination measures. By contrast, we assume the measure to be provided by the programmer and use it to prove finite blocking. Combining our work with inference techniques is an interesting direction for future work, especially in the presence of credits.

Deadlock freedom. There are numerous verification techniques and type systems to check deadlock freedom of programs that either synchronize via locks [7, 11, 24] or communicate via messages [5, 10]. Our work adopts Chalice's solution to checking deadlock freedom, and we refer to Leino et al. [15] for a detailed comparison to related work. The contribution of our work is to recast the Chalice solution in a uniform framework that supports a variety of blocking operations and to fix the soundness issue in Chalice that we mentioned above.

8 Conclusion

This paper introduces a novel verification technique to prove finite blocking in concurrent programs. At its core is a general framework for obligations, which express that a thread must perform a certain operation eventually. We present uniform proof rules for the manipulation of obligations and use them to encode three common blocking operations, which are representative for the various characteristics of obligations. By associating obligations with measures, our technique guarantees finite blocking even for programs containing non-

terminating threads under the assumption that scheduling, locks, and message receipt are strongly fair. Our technique subsumes termination checking and integrates verification of deadlock freedom.

As future work, we plan to use additional kinds of obligations to remove the main limitations or our technique. For instance, one could allow sending obligations over channels by introducing another form of obligation to ensure that every sent message will eventually be received such that the contained obligations do not get lost. Obligations to establish conditions on shared state could be used to prove that the busy-wait loop of a thread terminates. However, obligations are not limited to finite blocking. We plan to use the framework introduced here to prove other liveness properties, for instance, that every asynchronous task will be awaited eventually or that certain objects will be de-allocated eventually. Another direction for future work is to add support for abstract predicates [20] in order to denote statically-unknown sets of obligations in specifications, and to support information hiding. Finally, it would also be interesting to combine our work with approaches to infer termination measures.

Acknowledgments. We would like to Alex Summers for various discussions. We are grateful to the anonymous reviewers for their valuable comments. Pontus Boström was partially funded by a scholarship from Svenska kulturfonden. Peter Müller's work was funded in part by the Hasler Foundation.

References

1. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.
2. B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *POPL*, pages 265–276. ACM, 2007.
3. B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *PLDI*, pages 320–330. ACM, 2007.
4. B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *Commun. ACM*, 54(5), 2011.
5. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In D. Thomas, editor, *ECOOP*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.
6. P. Ganty and S. Genaim. Proving termination starting from the end. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 397–412. Springer, 2013.
7. C. S. Gordon, M. D. Ernst, and D. Grossman. Static lock capabilities for deadlock freedom. In *TLDI*, pages 67–78. ACM, 2012.
8. A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL*, pages 16–28. ACM, 2009.
9. G. J. Holzmann. *SPIN Model Checker, The: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
10. N. Kobayashi. A new type system for deadlock-free processes. In C. Baier and H. Hermanns, editors, *CONCUR*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.
11. D.-K. Le, W.-N. Chin, and Y.-M. Teo. An expressive framework for verifying deadlock freedom. In D. Van Hung and M. Ogawa, editors, *ATVA*, volume 8172 of *LNCS*, pages 287–302. Springer, 2013.

12 T. C. Le, C. Gherghina, A. Hobor, and W.-N. Chin. A resource-based logic for termination and non-termination proofs. In S. Merz and J. Pang, editors, *ICFEM*, volume 8829 of *LNCS*, pages 267–283. Springer, 2014.
13 K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *LPAR*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
14 K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *ESOP*, volume 5502 of *LNCS*, pages 378–393. Springer, 2009.
15 K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In A. D. Gordon, editor, *ESOP*, volume 6012 of *LNCS*, pages 407–426. Springer, 2010.
16 Z. Manna and A. Pnueli. A hierarchy of temporal properties. In C. Dwork, editor, *PODC*, pages 377–410. ACM, 1990.
17 Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–130, 1991.
18 Z. Manna and A. Pnueli. Verification of parameterized programs. *Specification and Validation Methods*, pages 167–230, 1995.
19 M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455. ACM, 2007.
20 M. J. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM, 2005.
21 M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. In G. Barthe, editor, *ESOP*, volume 6602 of *LNCS*, pages 439–458. Springer, 2011.
22 J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society Press, 2002.
23 J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In S. Drossopoulou, editor, *ECOOP*, volume 5653 of *LNCS*, pages 148–172. Springer, 2009.
24 K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In G. Ramalingam, editor, *APLAS*, volume 5356 of *LNCS*, pages 155–170. Springer, 2008.

A Credits and Deadlock Freedom

As explained in Sec. 2.1, blocking operations where the very first execution blocks (such as receiving on a channel) are handled by credits. Creating a credit simultaneously creates a corresponding obligation. Therefore, by enforcing that a thread executing a receive statement holds a sends-credit, we ensure that some other thread has a sends-obligation and, thus, the receive will not block indefinitely.

The producer-consumer example in Fig. 15 demonstrates this idea. The consumer method `Cons` requires a sends-credit for the channel `c`, which allows it to receive one message on this channel. Because of `Cons`'s precondition, this initial credit is provided by the `Main` method when it forks the consumer, which leaves the corresponding sends-obligation in `Main`. This obligation is transferred to the producer when the `Prod` method is forked. Note that `Main` could not terminate without forking the producer first because it would still hold an obligation and, thus, not pass the leak check at the end of the method. Note further that neither the producer nor the consumer promise to terminate and, thus, cannot be joined since the join operation might block indefinitely.

Once the producer and consumer have been forked, they communicate via the channel `c`. The declaration of `c`'s type `C` specifies that messages sent over the channel are boolean values.

```
channel C(b: bool) where b ⇒ sends(this, -1, ⊤);
```

```
method Main() {
  c := new C;
  fork t1 := Cons(c) below c;
  fork t2 := Prod(c);
}

method Prod(c: C)
  requires sends(ch, 1, 1);
{
  while(*)
    invariant sends(c, 1, 1);
  { send c(true); }
  send c(false);
}
```

```
method Cons(c: C)
  requires sends(c, -1, ⊤)
  requires waitlevel ≪ c;
{
  more := true;
  while(more)
    invariant more ⇒ sends(c, -1, 1);
    invariant waitlevel ≪ c;
  { receive more := c; }
}
```

Figure 15 A producer-consumer example. The producer and consumer communicate over an asynchronous channel `c`. The main method transfers a sends-credit to the consumer, which allows it to receive the first message, and the corresponding sends-obligation to the producer, forcing it to send a message. With every message except the final one, the producer sends another sends-credit to the consumer, which allows the consumer to receive the next message. The measure $\top$ is explained in Sec. 3.2.

Its channel invariant expresses that whenever the value true is sent over the channel, the message includes one sends-credit for the channel. Therefore, with every send operation inside the while loop of method `Prod`, the producer sends a credit to the consumer. Consequently, the producer has one sends-obligation throughout the loop because it satisfies one obligation by sending a message and obtains a new one by sending away a credit. This property is expressed by its loop invariant. Since the sends-obligation gets satisfied in each loop iteration, its measure is constant 1. However, similar examples require other measures; for instance, if the producer sent messages to several channels in a round-robin fashion, the sends-obligation for each of the channels would be the number of channels. Once the loop has terminated, the producer sends a final message not containing a credit. This send operation satisfies the remaining sends-obligation, allowing method `Prod` to pass its leak check and terminate. The consumer obtains another credit with every message it receives, which allows it to receive the next message. The final message (with value false) contains no credit, forcing the consumer to terminate its receive-loop.

To prevent deadlock, the receive operation in the consumer requires that the consumer's wait level is strictly below the level of the channel c. This constraint is required in the precondition and maintained throughout the loop. In order to satisfy the precondition, method `Main` forks the consumer with an initial wait level that is below c's level (indicated by the `below`-clause). We omit such constraints from our encoding, but their treatment is straightforward [15].

Modular Termination Verification

Bart Jacobs[1], Dragan Bosnacki[2], and Ruurd Kuiper[2]

1 **iMinds-DistriNet, Department of Computer Science, KU Leuven, Belgium**
`bart.jacobs@cs.kuleuven.be`
2 **Eindhoven University of Technology, The Netherlands**
`{d.bosnacki,r.kuiper}@tue.nl`

Abstract

We propose an approach for the modular specification and verification of total correctness properties of object-oriented programs. We start from an existing program logic for partial correctness based on separation logic and abstract predicate families. We extend it with *call permissions* qualified by an arbitrary ordinal number, and we define a specification style that properly hides implementation details, based on the ideas of using methods and bags of methods as ordinals, and exposing the bag of methods reachable from an object as an abstract predicate argument. These enable each method to abstractly request permission to call all methods reachable by it any finite number of times, and to delegate similar permissions to its callees. We illustrate the approach with several examples.

1998 ACM Subject Classification F.3.1 Logics and Meanings of Programs

Keywords and phrases Termination, program verification, modular verification, separation logic, well-founded relations

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.664

1 Introduction

Software plays a significant role in ever more areas of human activity, and in ever more applications with high reliability requirements, where failures caused by software defects could affect human safety, system security, or mission success. In many cases, software verification through testing provides insufficient assurance of the absence of defects. Formal program verification, where the program's source code is analyzed to obtain mathematical certainty that all of a program's possible executions satisfy certain formalized requirements, is in such cases a promising complementary approach.

Formal program verification approaches can be roughly divided into two categories: *whole-program* approaches and *modular* approaches. In a whole-program approach, a complete, closed program must be available before any results can be obtained. In such an approach, a method call is verified by verifying the method's implementation, taking into account the particular context of the call. If a call is dynamically bound, all potential callees are inspected. A major advantage of a whole-program approach is that typically, besides the source code itself and a formalization of the overall correctness property being verified, little or no additional user input is required. A disadvantage is that modifying any part of the program invalidates the results obtained.

In a modular approach, on the other hand, the object of verification is not whole programs, but program *modules*, coherent sets of classes and interfaces, developed independently, that satisfy a well-defined *module specification*. A module need not be *closed*: it may refer to classes and interfaces not defined by the module itself, but by other modules which it *imports*. A module should use only those elements (classes, interfaces, methods) from an

© Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 664–688

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

imported module that are specified as *exported* (or *public*) by that module's specification; only those elements are guaranteed to still be present in future versions of the imported module. *Verifying* a module means proving that it satisfies its specification, assuming that imported modules satisfy theirs.

In a modular approach, a method call is verified by assuming that it satisfies the called method's specification. If the call is dynamically bound, only the specification of the statically resolved method is considered. Separately, it is checked that each overriding method satisfies the specification of each method it overrides. In this approach, after modifying a method, it is sufficient to check that the method still satisfies its specification, to ensure that any properties verified previously still hold.

A major issue in modular verification is the question of the *specification approach*: what should a module specification look like? The approach should be sufficiently *expressive* to be able to capture precisely the dependencies that a module's clients (i.e. those other modules that import the module) may have upon it, but it should also be sufficiently *abstract* so that proper *information hiding* is achieved: a module's specification should not unnecessarily constrain the current implementation or its future evolution. Any modification that does not break clients should be allowed.

In recent years, great progress has been made in specification approaches for *partial correctness*, the property that the program never reaches a incorrect state. However, we are not aware of any existing approach for modular specification of *total correctness* of object-oriented programs, the property that additionally the program *terminates*.

In this paper, we propose such an approach.

For sequential programs, termination means absence of infinite loops and absence of infinite recursion. In the remainder of this paper, we assume the program has no loops; this can be achieved by turning each loop into a recursive method.

The main difficulty in defining a specification approach for modular verification of termination of object-oriented programs, is in dealing with dynamic binding. This can be understood as follows. Firstly, we assume that the module import graph is acyclic. (That is, no module directly or indirectly imports itself. If there is such a cycle, its members should be consolidated into a single module.) This assumption allows us to think of the program as consisting of *layers*, such that modules only import modules from lower layers. In the absence of dynamic binding, all method calls are either internal within a module, or descend into a lower layer. (Indeed, a module should refer only to the classes and interfaces it defines itself and the ones it imports.) Therefore, any cycle in the call graph is necessarily internal to a module, so proving absence of infinite recursion is not directly a module specification issue. Indeed, the number of non-intra-module calls in a call stack below a given call is bounded by the *static depth* of the caller, i.e. the number of layers below it.

In contrast, in the presence of dynamic binding, if we define an intra-module call as a call where the caller module knows statically (based on its specification and the specifications of the modules it imports) that the callee is internal to itself, then the number of non-intra-module calls in a chain of calls is not bounded, so absence of infinite intra-module recursion does not imply absence of infinite recursion. This is the main problem addressed in this paper.

For example, consider the program of Fig. 1. In this program, we can consider interface `RealFunc` as well as classes `Math`, `Identity`, and `Loopy` to each constitute a separate module, where `Math` imports `RealFunc`, and `Identity` and `Loopy` each import both `RealFunc` and `Math`. Notice that none of these modules contain intra-module method calls; nonetheless, whereas method `Identity.test` correctly returns the derivative of the identity function at argument 42,

```
interface RealFunc {
  float apply(float x);
}
class Math {
  static float derivative(RealFunc f, float x)
  { f.apply(x + 1) − f.apply(x) }
}
```

```
class Identity implements RealFunc {
  float apply(float x) { x }
  static float test()
  { Math.derivative(new Identity(), 42) }
}
class Loopy implements RealFunc {
  float apply(float x)
  { Math.derivative(this, x) }
  static float test()
  { Math.derivative(new Loopy(), 42) }
}
```

Figure 1 An example program without intra-module recursion but with infinite inter-module recursion.

method Loopy.test performs infinite inter-module recursion between methods Loopy.apply and Math.derivative.

In this paper, we propose:

- a *program logic* for expressing module specifications that specify total correctness properties of methods exported by these modules, such as termination of method Identity.test;
- a corresponding *proof system* for verifying modules against their specifications that is *sound*, i.e. if a proof exists in this proof system for each module of a program, then each method satisfies its specified total correctness properties, implying that the proof system does not allow the verification of a specification that states that method Loopy.test terminates; also, the proof system is *modular*, meaning that each module's proof uses only the *specifications*, not the *implementations* of imported modules, and does not depend at all on other modules, such that the proof of Math.derivative uses only the specification of module RealFunc, and does not depend on the existence or non-existence, or the content, of modules Identity and Loopy, and the proof of Identity.test uses only the specification of Math.derivative and not its implementation; and
- a *specification style* for writing specifications in this program logic such that they perform proper information hiding, e.g. such that method Math.derivative's specification is satisfied equally by alternative implementations that are more complex (and more accurate).

Our approach is based on the observation that any dynamically bound call is a call on an *object*. This object, together with the objects reachable from it via field dereferences, constitutes a *data structure*. At any point during program execution, the data structures existing in memory at that point are of finite size, and were composed of objects of classes from different modules in a finite number of composition steps. The core idea of our approach, then, is to associate with each data structure, at each point in time, a *dynamic depth*. This is roughly the number of objects in the data structure. More precisely, to allow each module, even while operating on a data structure, to create new data structures composed from classes in lower-layer modules and perform calls on them, we track each object's module, so the dynamic depth is (more or less) the multiset of modules contributing objects to the data structure. By using these dynamic depths as part of a recursion measure, we obtain a specification style that performs proper information hiding.

The rest of this paper is structured as follows. To define our approach precisely, we start from an existing modular specification and verification approach for partial correctness of

object-oriented programs, based on *separation logic* to deal with aliased mutable memory and *abstract predicate families* to achieve properly abstract specifications. We recall this existing approach in Sec. 2. In Sec. 3, we extend the program logic of this partial correctness approach with *call permissions* to obtain a program logic for total correctness. This logic is based on the well-known tools for reasoning about termination, *well-founded relations* and *ordinal numbers*. However, this logic is not the main contribution of the paper. The main issue addressed in this paper is: how to use this logic to write module specifications that are both *expressive* and *abstract*? In Sec. 4, in three steps we build up our modular specification approach, and we illustrate and motivate it through a sequence of examples. In Sec. 5, we briefly discuss how we added support for our approach to our program verification tool VeriFast. We discuss related work in Sec. 6 and we conclude in Sec. 7.

2 Separation logic and abstract predicate families

We start from an existing approach for modular specification and verification of partial correctness properties of object-oriented programs, based on separation logic [9, 2] and abstract predicate families [10]. We introduce the approach informally in Sec. 2.1. We formally define the approach in Sec. 2.2.

2.1 An Example

We present our approach in the context of a simple Java-like programming language. Consider the program in Fig. 4. It defines an interface IntFunc, two classes, PlusN and Twice, that implement the interface, and a class Program with a method main that composes a complex object and performs a dynamically bound call on it. An object **new** PlusN(y) represents a function that takes an integer x and returns the value $x + y$. An object **new** Twice(f), where f is itself an IntFunc object, represents a function that maps a value x to $f(f(x))$.

Annotations, which have no effect on the run-time behavior of the program and serve only for modular specification and verification, are shown on a gray background. Besides the presence of annotations, the main differences of our programming language with Java are inspired by Scala: fields are declared in a parenthesized list after the class name instead of in the body of the class; **new** expressions specify an initial value for each field; and the final expression in a method body determines the return value, without the need for a **return** keyword. (Note: unlike in Scala, field names are not in scope in the class body; to access a field f, one must write this.f.)

To enable modular verification, each method has a *contract*, i.e. a specification consisting of a precondition and a postcondition (prefixed by keywords **req** and **ens**, respectively). Note, however, that no contract is declared explicitly for methods apply of classes PlusN and Twice; they inherit their contract from method apply of interface IntFunc, which they override.

While this example program does not perform heap mutation (i.e. it does not modify any fields of any objects), our approach supports this fully. Therefore, method specifications should specify not only what should be true on entry to the method and what should be true on exit from the method, but also which object fields are modified by the method, and which are not. This is known as the *frame condition*. For this purpose we use *separation logic*. One way to understand separation logic, when applied to a Java-like programming language with garbage collection, is by saying that it drops the assumption that reachable objects are allocated. Furthermore, we drop the assumption that all fields of a given object are allocated together. When verifying a program using separation logic, we need to prove

that whenever the program accesses a field, that field is allocated. As a result, a method's precondition needs to state which fields it expects to be allocated. Since a verified method accesses only fields whose allocatedness is asserted by its precondition, we can infer that any fields that are allocated at a given call site but whose allocatedness is not asserted by the callee's precondition, remain unchanged by the call.

For example, consider the following example program:

```
class Account(int balance) {
  static void transfer(Account from, Account to, int amount)
    req from.balance ↦ b1 ∗ to.balance ↦ b2;
    ens from.balance ↦ b1 − amount ∗ to.balance ↦ b2 + amount;
  {
    int bal1 := from.balance; from.balance := bal1 − amount;
    int bal2 := to.balance; to.balance := bal2 + amount;
  }
}
```

This program transfers an amount of money between two bank accounts. The separation logic assertion from.balance $\mapsto$ b1 asserts that field balance of object from is allocated and has value b1. Such a *points-to assertion* is the only way in separation logic assertions to specify the value of a field. This way, it is syntactically enforced that a separation logic assertion does not refer to the value of unallocated fields.

A *separating conjunction* $P * Q$, where P and Q are separation logic assertions, asserts that the heap (i.e. the set of allocated fields, with their values) can be split into two disjoint parts (i.e. where all of the fields that are allocated in one part are not allocated in the other part) such that P holds for one part, and Q holds for the other part. Therefore, it follows from the precondition of transfer that from and to are not the same object.

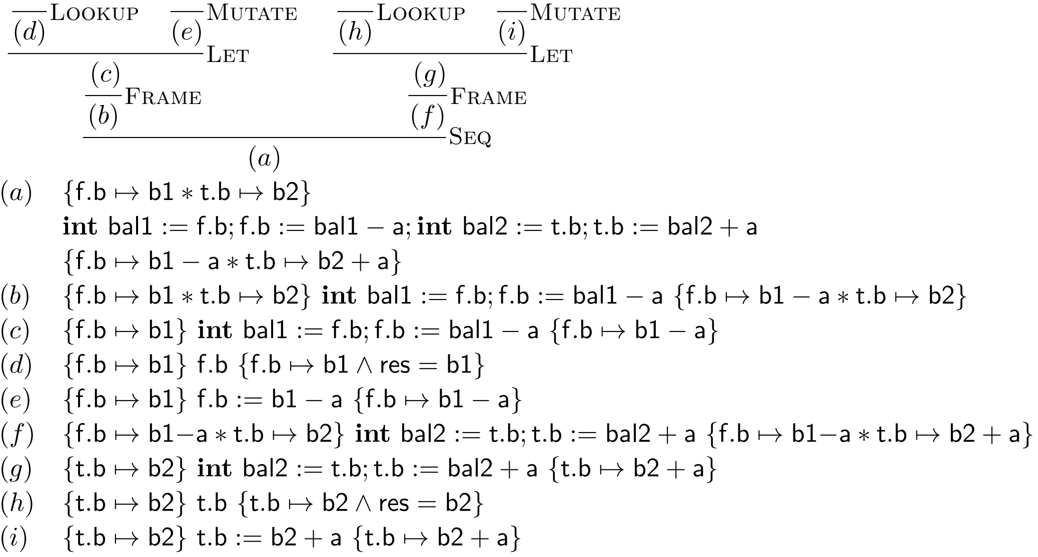
Notice that the variables b1 and b2 are free variables of the contract of transfer. The meaning of such free variables in this paper is as follows: free variables of the precondition are implicitly universally quantified at the level of the contract; their scope extends to the postcondition as well. That is, method transfer should satisfy its contract for all possible values of b1 and b2. (Variables that are free only in the postcondition are existentially quantified at the level of the postcondition; see the examples later in this paper.)

Verifying a method with precondition P, postcondition Q, and body B, means proving the *Hoare triple* $\vdash \{P\}\ B\ \{Q\}$ using the *proof rules* of the program logic. The proof rules of the logic of this section are shown in Fig. 7. Notice that the precondition of proof rule Mutate mentions only the field being modified. Information about other fields can be preserved using rule Frame. Notice also that there is no proof rule for sequential composition. Indeed, we treat a sequential composition $c; c'$ as a shorthand for a **let** command $\tau\ x = c; c'$, for some arbitrary type τ and some fresh variable x. As a result, we can derive the usual proof rule for sequential composition:

$$\textsc{Seq}\quad \frac{\vdash \{P\}\ c\ \{Q\} \qquad \vdash \{Q\}\ c'\ \{R\}}{\vdash \{P\}\ c; c'\ \{R\}}$$

A proof tree for method transfer is shown in Fig. 2. We abbreviated the identifiers in obvious ways. Notice that we treat assertions semantically; e.g. since the assertions $\mathsf{f.b} \mapsto \mathsf{b1}{-}\mathsf{a} * \mathsf{t.b} \mapsto \mathsf{b2}$ and $\mathsf{t.b} \mapsto \mathsf{b2} * \mathsf{f.b} \mapsto \mathsf{b1}{-}\mathsf{a}$ are equivalent, we treat them as equal.

We show the same proof tree in the more convenient form of a *proof outline* in Fig. 3.



(a) $\{\mathsf{f.b} \mapsto \mathsf{b1} * \mathsf{t.b} \mapsto \mathsf{b2}\}$
$\quad$ $\mathbf{int}\ \mathsf{bal1} := \mathsf{f.b}; \mathsf{f.b} := \mathsf{bal1} - \mathsf{a}; \mathbf{int}\ \mathsf{bal2} := \mathsf{t.b}; \mathsf{t.b} := \mathsf{bal2} + \mathsf{a}$
$\quad$ $\{\mathsf{f.b} \mapsto \mathsf{b1} - \mathsf{a} * \mathsf{t.b} \mapsto \mathsf{b2} + \mathsf{a}\}$

(b) $\{\mathsf{f.b} \mapsto \mathsf{b1} * \mathsf{t.b} \mapsto \mathsf{b2}\}\ \mathbf{int}\ \mathsf{bal1} := \mathsf{f.b}; \mathsf{f.b} := \mathsf{bal1} - \mathsf{a}\ \{\mathsf{f.b} \mapsto \mathsf{b1} - \mathsf{a} * \mathsf{t.b} \mapsto \mathsf{b2}\}$

(c) $\{\mathsf{f.b} \mapsto \mathsf{b1}\}\ \mathbf{int}\ \mathsf{bal1} := \mathsf{f.b}; \mathsf{f.b} := \mathsf{bal1} - \mathsf{a}\ \{\mathsf{f.b} \mapsto \mathsf{b1} - \mathsf{a}\}$

(d) $\{\mathsf{f.b} \mapsto \mathsf{b1}\}\ \mathsf{f.b}\ \{\mathsf{f.b} \mapsto \mathsf{b1} \wedge \mathsf{res} = \mathsf{b1}\}$

(e) $\{\mathsf{f.b} \mapsto \mathsf{b1}\}\ \mathsf{f.b} := \mathsf{b1} - \mathsf{a}\ \{\mathsf{f.b} \mapsto \mathsf{b1} - \mathsf{a}\}$

(f) $\{\mathsf{f.b} \mapsto \mathsf{b1}{-}\mathsf{a} * \mathsf{t.b} \mapsto \mathsf{b2}\}\ \mathbf{int}\ \mathsf{bal2} := \mathsf{t.b}; \mathsf{t.b} := \mathsf{bal2} + \mathsf{a}\ \{\mathsf{f.b} \mapsto \mathsf{b1}{-}\mathsf{a} * \mathsf{t.b} \mapsto \mathsf{b2} + \mathsf{a}\}$

(g) $\{\mathsf{t.b} \mapsto \mathsf{b2}\}\ \mathbf{int}\ \mathsf{bal2} := \mathsf{t.b}; \mathsf{t.b} := \mathsf{bal2} + \mathsf{a}\ \{\mathsf{t.b} \mapsto \mathsf{b2} + \mathsf{a}\}$

(h) $\{\mathsf{t.b} \mapsto \mathsf{b2}\}\ \mathsf{t.b}\ \{\mathsf{t.b} \mapsto \mathsf{b2} \wedge \mathsf{res} = \mathsf{b2}\}$

(i) $\{\mathsf{t.b} \mapsto \mathsf{b2}\}\ \mathsf{t.b} := \mathsf{b2} + \mathsf{a}\ \{\mathsf{t.b} \mapsto \mathsf{b2} + \mathsf{a}\}$

Figure 2 Proof tree for method transfer.

$\{\mathsf{f.b} \mapsto \mathsf{b1} * \mathsf{t.b} \mapsto \mathsf{b2}\}$
$\quad\{\mathsf{f.b} \mapsto \mathsf{b1}\}$ $\quad$ Frame
$\quad\mathbf{int}\ \mathsf{bal1} := \mathsf{f.b};$
$\quad\{\mathsf{f.b} \mapsto \mathsf{b1} \wedge \mathsf{bal1} = \mathsf{b1}\}$
$\quad\mathsf{f.b} := \mathsf{bal1} - \mathsf{a};$
$\quad\{\mathsf{f.b} \mapsto \mathsf{b1} - \mathsf{a}\}$
$\{\mathsf{f.b} \mapsto \mathsf{b1} - \mathsf{a} * \mathsf{t.b} \mapsto \mathsf{b2}\}$
$\quad\{\mathsf{t.b} \mapsto \mathsf{b2}\}$ $\quad$ Frame
$\quad\mathbf{int}\ \mathsf{bal2} := \mathsf{t.b};$
$\quad\{\mathsf{t.b} \mapsto \mathsf{b2} \wedge \mathsf{bal2} = \mathsf{b2}\}$
$\quad\mathsf{t.b} := \mathsf{bal2} + \mathsf{a}$
$\quad\{\mathsf{t.b} \mapsto \mathsf{b2} + \mathsf{a}\}$
$\{\mathsf{f.b} \mapsto \mathsf{b1} - \mathsf{a} * \mathsf{t.b} \mapsto \mathsf{b2} + \mathsf{a}\}$

Figure 3 Proof outline for method transfer.

By the soundness of the program logic, the fact that we succeeded in proving this Hoare triple implies that every execution of method transfer that starts in a state that satisfies the precondition (for certain values of b1 and b2) will not access unallocated memory and, if it terminates, its final state satisfies the postcondition (for the same values of b1 and b2).

Returning now to the example program of Fig. 4, we see that method apply of class PlusN accesses field this.y; therefore, its precondition should assert that this field is allocated. However, since method apply overrides the corresponding method of interface IntFunc, we have to conclude that the precondition of method apply in interface IntFunc should assert that this.y is allocated. Clearly, it would not make sense for the interface method's contract to assert this directly. Rather, at the level of the interface, method apply's precondition should assert abstractly that whatever fields belong to the object's representation should be allocated. The specification approach supports this by means of *abstract predicate families*. An *abstract predicate* is simply a named and possibly parameterized separation logic assertion. An *abstract predicate family* is an abstract predicate declared at the level of an interface, and defined at the level of each of the classes that implement the interface. Predicate IntFunc declared in interface IntFunc is such an abstract predicate family.[1] Its intended meaning at the level of interface IntFunc is that it asserts the allocatedness of the fields belonging to the IntFunc object, as well as any validity constraints over their values. It corresponds to what is known as a *class invariant* in some other modular verification approaches. It is then natural that method apply asserts this predicate in its precondition and in its postcondition.

Notice that class PlusN defines this predicate to assert allocatedness of field this.y. (The underscore denotes existential quantification of the field value; i.e. the assertion does not assert anything about the field value.) Therefore, method apply of class PlusN verifies.

Method apply of class Twice calls the apply method of the object pointed to by its f field. Therefore, the definition of predicate IntFunc at the level of class Twice asserts not only the allocatedness of field this.f, but also the predicate f.IntFunc.[2]

2.2 Formal Definition

We formally define the program logic for partial correctness that we start from.

The syntax of the programming language and the annotations is shown in Fig. 5.

We assume a set of interface names $\iota \in \mathit{ItfNames}$ and a set of class names $C \in \mathit{ClassNames}$. The types τ of the programming language include at least the types **int** of integers and **bool** of booleans, and the interface types ι and class types C. Correspondingly, the values $v \in \mathit{Values}$ of the programming language include at least the integers $z \in \mathbb{Z}$ and the booleans $b \in \mathbb{B}$, and the object references $o \in \mathit{ObjRefs}$. We will assume additional types and values whenever useful for particular examples.

The expressions include the literal values v, the variable references x, and the pure operations $op(\overline{e})$ that map a sequence of argument values to a result value. The separation logic assertions P include the boolean expressions e, asserting that the expression evaluates to true; the points-to assertions $e.f \mapsto e$, asserting that the indicated field is present in the

[1] In this paper we adopt the convention of using the name of the interface also as the name of its abstract predicate family (when it declares exactly one abstract predicate family, which is usually the case); however, this is an arbitrary choice. Another reasonable name would be valid.

[2] Note that such a recursive reference to predicate IntFunc inside a definition of IntFunc never causes well-definedness problems, provided that each reference to a predicate inside a predicate definition is in a *positive position*, i.e. not underneath a negation or on the left-hand side of an implication; in that case, the set of predicate definitions of a program, seen as a system of equations, always has a solution. See also Sec. 2.2.

```
class PlusN(int y) implements IntFunc {
  predicate IntFunc() = this.y ↦ _;
  int apply(int x)
  { int y := this.y; x + y }
  static IntFunc createPlusN(int y)
    req true; ens result.IntFunc();
  { new PlusN(y) }
}
class Program {
  static void main()
    req true; ens true;
  {
    IntFunc f1 := PlusN.createPlusN(10);
    IntFunc f2 := Twice.createTwice(f1);
    IntFunc f3 := Twice.createTwice(f2);
    f3.apply(42)
  }
}
```

```
interface IntFunc {
  predicate IntFunc();
  int apply(int x);
    req this.IntFunc();
    ens this.IntFunc();
}
class Twice(IntFunc f)
    implements IntFunc {
  predicate IntFunc() =
    this.f ↦ f ∗ f.IntFunc();
  int apply(int x) {
    IntFunc f := this.f;
    int y := f.apply(x);
    f.apply(y)
  }
  static IntFunc createTwice(IntFunc f)
    req f.IntFunc();
    ens result.IntFunc();
  { new Twice(f) }
}
```

Figure 4 Example program annotated with partial correctness specifications.

$$
\begin{array}{rcl}
\tau & ::= & \textbf{int} \mid \textbf{bool} \mid \iota \mid C \mid \cdots \\
e & ::= & v \mid x \mid op(\overline{e}) \\
P & ::= & e \mid e.f \mapsto e \mid P * P \mid P \wedge P \mid P \vee P \mid e.p(\overline{e}) \\
c & ::= & e \mid \tau\ x := c; c \mid \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c \mid \{\ c\ \} \\
 & & \mid C.m(\overline{e}) \mid e.m(\overline{e}) \mid \textbf{new}\ C(\overline{e}) \mid e.f \mid e.f := e \\
\mathit{pdecl} & ::= & \textbf{predicate}\ p(\overline{\tau\ x}); \\
\mathit{pdef} & ::= & \textbf{predicate}\ p(\overline{\tau\ x}) = P; \\
\mathit{imdef} & ::= & \tau\ m(\overline{\tau\ x});\ \ \textbf{req}\ P;\ \textbf{ens}\ P; \\
\mathit{mkind} & ::= & \textbf{static} \mid \textbf{instance} \\
\mathit{cmdef} & ::= & \mathit{mkind}\ \tau\ m(\overline{\tau\ x})\ \ \textbf{req}\ P;\ \textbf{ens}\ P;\ \ \{\ c\ \} \\
\mathit{idef} & ::= & \textbf{interface}\ \iota\ \{\ \overline{\mathit{pdecl}}\ \ \overline{\mathit{imdef}}\ \} \\
\mathit{cdef} & ::= & \textbf{class}\ C(\overline{\tau\ f})\ \textbf{implements}\ \iota\ \{\ \overline{\mathit{pdef}}\ \ \overline{\mathit{cmdef}}\ \} \\
\mathit{tdef} & ::= & \mathit{idef} \mid \mathit{cdef} \\
\mathit{program} & ::= & \overline{\mathit{tdef}}
\end{array}
$$

Figure 5 Syntax of the programming language and the annotations.

heap and holds the indicated value; the separating conjunction $P * P$, asserting that the heap can be split into two parts such that one conjunct holds in one part, and the other conjunct holds in the other part; regular conjunction and disjunction; and *predicate assertions* $e.p(\overline{e})$, asserting that the indicated predicate holds with the indicated argument values. p ranges over predicate names.

Like expressions, commands c return a value; unlike expressions, they may also access the heap and have side-effects. The commands include the expressions; a **let**-like construct $\tau\ x := c; c'$ that first executes c, binds the result to variable x of type τ, and then executes c'; conditional commands; parenthesized commands; static and instance method calls; object creation, field lookup, and field mutation commands.

A predicate declaration specifies a predicate name and a parameter list; a predicate definition additionally specifies a body.

An interface method specifies a return type, a method name, a parameter list, and a contract consisting of a precondition and a postcondition. A class method additionally specifies a kind (kind **instance** is the default and is usually left implicit) and a body.

An interface definition declares a number of predicate families and a number of interface methods. A class definition declares a list of fields (empty if omitted), an implemented interface (interface Empty, that declares no predicate families and no methods, if omitted), a number of predicate family instances, and a number of class methods.

The type definitions are the interface definitions and the class definitions. A program is a sequence of type definitions.

We assume a function $\mathsf{classOf} : \mathit{ObjRefs} \to \mathit{ClassNames}$ such that infinitely many object references map to any given class. A heap $h \in \mathit{Heaps} = \mathit{ObjRefs} \times \mathit{FieldNames} \rightharpoonup \mathit{Values}$ is a partial function from pairs of object references and field names to values. We do not allow instantiation of classes that have no fields; therefore, the set of allocated objects can be derived from $\mathrm{dom}(h)$.

We define the semantics of programs by means of a big-step relation $(h, c) \Downarrow \gamma$ that relates a pre-heap and a closed command (i.e. a command with no free variables) to an outcome γ, which is either of the form (n, v, h') where $n \in \mathbb{N}$ is the number of execution steps performed, v is the result value, and h' is the post-heap, or an *exception* E, which is either $\mathbf{Failure}(n)$, where $n \in \mathbb{N}$ is the number of execution steps performed, or **Divergence**. We define $n + \gamma$ as follows: $n + (n', v, h') = (n + n', v, h')$; $n + \mathbf{Failure}(n') = \mathbf{Failure}(n + n')$; $n + \mathbf{Divergence} = \mathbf{Divergence}$. We define the big-step relation coinductively [8] by means of the rules shown in Fig. 6.

Note that $h \uplus h'$ is undefined if $\mathrm{dom}(h) \cap \mathrm{dom}(h') \neq \emptyset$.

We now define the meaning of assertions. To interpret an assertion, we need an interpretation for the predicates it uses. A predicate interpretation I is a set $I \subseteq \mathit{ObjRefs} \times \mathit{PredNames} \times \mathit{Values}^* \times \mathit{Heaps}$. If $(o, p, \overline{v}, h) \in I$, this means that according to interpretation I, predicate assertion $o.p(\overline{v})$ is true in heap h. We now define the truth $I, h \vDash P$ of a closed assertion P under a predicate interpretation I and a heap h:

$$\begin{array}{lcl}
I, h \vDash v & \Leftrightarrow & v = \mathsf{true} \\
I, h \vDash o.f \mapsto v & \Leftrightarrow & (o, f) \mapsto v \in h \\
I, h \vDash P * P' & \Leftrightarrow & \exists h_1, h_2.\ h = h_1 \uplus h_2 \land I, h_1 \vDash P \land I, h_2 \vDash P' \\
I, h \vDash P \land P' & \Leftrightarrow & I, h \vDash P \land I, h \vDash P' \\
I, h \vDash P \lor P' & \Leftrightarrow & I, h \vDash P \lor I, h \vDash P' \\
I, h \vDash o.p(\overline{v}) & \Leftrightarrow & (o, p, \overline{v}, h) \in I
\end{array}$$

Given a predicate interpretation I, we can interpret the predicate definitions of a program to

$$\gamma ::= (n, v, h) \mid E \qquad E ::= \textbf{Failure}(n) \mid \textbf{Divergence} \qquad (h, v) \Downarrow (1, v, h)$$

$$\frac{(h, c) \Downarrow (n, v, h') \qquad (h', c'[v/x]) \Downarrow \gamma}{(h, \tau\ x := c; c') \Downarrow n + \gamma} \qquad \frac{(h, c) \Downarrow E}{(h, \tau\ x := c; c') \Downarrow 1 + E}$$

$$\frac{\textbf{class}\ C \cdots \{\ \cdots\ \textbf{static}\ \tau\ m(\overline{\tau\ x})\ \{\ c\ \}\ \cdots\ \} \qquad (h, c[\overline{v}/\overline{x}]) \Downarrow \gamma}{(h, C.m(\overline{v})) \Downarrow 1 + \gamma}$$

$$\frac{\mathsf{classOf}(o) = C \quad \textbf{class}\ C \cdots \{\ \cdots\ \textbf{instance}\ \tau\ m(\overline{\tau\ x})\ \{\ c\ \}\ \cdots\ \} \qquad (h, c[o, \overline{v}/\mathsf{this}, \overline{x}]) \Downarrow \gamma}{(h, o.m(\overline{v})) \Downarrow 1 + \gamma}$$

$$\frac{\mathsf{classOf}(o) = C \qquad \textbf{class}\ C(\overline{\tau\ f})\ \cdots \qquad h' = h \uplus \{\overline{o.f \mapsto v}\}}{(h, \textbf{new}\ C(\overline{v})) \Downarrow (1, o, h')} \qquad \frac{(o, f) \in \mathrm{dom}(h)}{(h, o.f) \Downarrow (1, h((o, f)), h)}$$

$$\frac{(o, f) \notin \mathrm{dom}(h)}{(h, o.f) \Downarrow \textbf{Failure}(1)} \qquad \frac{(o, f) \in \mathrm{dom}(h)}{(h, o.f := v) \Downarrow (1, v, h[(o, f) := v])} \qquad \frac{(o, f) \notin \mathrm{dom}(h)}{(h, o.f := v) \Downarrow \textbf{Failure}(1)}$$

Figure 6 Coinductive big-step semantics $(h, c) \Downarrow \gamma$ of the programming language.

obtain a new predicate interpretation $F(I)$:

$$\frac{\mathsf{classOf}(o) = C \quad \textbf{class}\ C \cdots \{\ \cdots\ \textbf{predicate}\ p(\overline{\tau\ x}) = P;\ \cdots\ \} \qquad \overline{y} = \mathsf{FV}(P[\overline{v}/\overline{x}]) \qquad I, h \vDash P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]}{(o, p, \overline{v}, h) \in F(I)}$$

Notice that free variables in a predicate body are implicitly existentially quantified.

It is easy to check that F is monotonic: $I \subseteq I' \Rightarrow F(I) \subseteq F(I')$. (This would not be the case if our assertion language included negation or implication of assertions.) Therefore, by the Knaster-Tarski theorem, $I_{\text{fix}} = \bigcap\{I \mid F(I) \subseteq I\}$ is the least fixpoint of F. We adopt I_{fix} as the meaning of predicates.

We are now ready to define the meaning of Hoare triples (for partial correctness):

$$\vDash \{P\}\ c\ \{Q\} \quad \Leftrightarrow \quad \forall h, \gamma.\ I_{\text{fix}}, h \vDash P \wedge (h, c) \Downarrow \gamma \Rightarrow \gamma \vDash Q$$

where satisfaction $\gamma \vDash Q$ of a postcondition by an outcome is defined as:

$$\textbf{Divergence} \vDash Q \qquad \frac{I_{\text{fix}}, h \vDash Q[v/\mathsf{res}]}{(n, v, h) \vDash Q}$$

The proof rules are shown in Fig. 7.

Notice that in method contracts, free variables in the precondition are universally quantified across the contract; their scope extends to the postcondition as well. Variables that are free only in the postcondition are existentially quantified in the postcondition.

We say that a class implements an interface method if the class has a method of the same name, return type, and parameter list, whose body satisfies the interface method's contract.

Expr
$$\vdash \{\mathsf{true}\}\ v\ \{\mathsf{res} = v\}$$

Let
$$\frac{\vdash \{P\}\ c\ \{Q\} \qquad \forall v.\ \vdash \{Q[v/\mathsf{res}]\}\ c'[v/x]\ \{R\}}{\vdash \{P\}\ \tau\ x := c; c'\ \{R\}}$$

StaticCall
$$\frac{\begin{array}{c}\textbf{class}\ C \cdots\ \{\ \cdots\ \textbf{static}\ \tau\ m(\overline{\tau\ x})\ \textbf{req}\ P;\ \textbf{ens}\ Q;\ \cdots\ \}\\ \overline{y} = \mathsf{FV}(P) \setminus \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \overline{x}, \mathsf{result}, \overline{y}\end{array}}{\vdash \{P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ C.m(\overline{v})\ \{\exists \overline{w'}.\ Q[\overline{v}/\overline{x}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\}}$$

DynamicCall
$$\frac{\begin{array}{c}\textbf{interface}\ \iota\ \{\ \cdots\ \tau\ m(\overline{\tau\ x});\ \textbf{req}\ P;\ \textbf{ens}\ Q;\ \cdots\ \}\\ \overline{y} = \mathsf{FV}(P) \setminus \mathsf{this}, \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \mathsf{this}, \overline{x}, \mathsf{result}, \overline{y}\end{array}}{\vdash \{P[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ o.m(\overline{v})\ \{\exists \overline{w'}.\ Q[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\}}$$

New
$$\frac{\textbf{class}\ C(\overline{\tau\ f})\ \cdots}{\vdash \{\mathsf{true}\}\ \textbf{new}\ C(\overline{v})\ \{\circledast_{(f,v)\in\overline{(f,v)}}\ \mathsf{res}.f \mapsto v\}}$$

Lookup
$$\vdash \{o.f \mapsto v\}\ o.f\ \{o.f \mapsto v \land \mathsf{res} = v\}$$

Mutate
$$\vdash \{o.f \mapsto _\}\ o.f := v\ \{o.f \mapsto v\}$$

Conseq
$$\frac{\vDash P \Rightarrow P' \qquad \vdash \{P'\}\ c\ \{Q'\} \qquad \vDash Q' \Rightarrow Q}{\vdash \{P\}\ c\ \{Q\}}$$

Frame
$$\frac{\vdash \{P\}\ c\ \{Q\}}{\vdash \{P * R\}\ c\ \{Q * R\}}$$

Disj
$$\frac{\vdash \{P\}\ c\ \{Q\} \qquad \vdash \{P'\}\ c\ \{Q\}}{\vdash \{P \lor P'\}\ c\ \{Q\}}$$

Program
$$\frac{\mathit{program} = \overline{\mathit{tdef}} \qquad \overline{\vdash \mathit{tdef}\ \mathsf{ok}}}{\vdash \mathit{program}\ \mathsf{ok}}$$

Interface
$$\vdash \mathit{idef}\ \mathsf{ok}$$

Class
$$\frac{\overline{C \vdash \mathit{cmdef}\ \mathsf{ok}} \qquad \textbf{interface}\ \iota\ \{\ \overline{\mathit{pdecl}}\ \overline{\mathit{imdef}}\ \} \qquad \overline{\vdash C\ \mathsf{implements}\ \mathit{imdef}}}{\vdash \textbf{class}\ C(\cdots)\ \textbf{implements}\ \iota\ \{\ \overline{\mathit{pdef}}\ \overline{\mathit{cmdef}}\ \}\ \mathsf{ok}}$$

StaticMethod
$$\frac{\begin{array}{c}\overline{y} = \mathsf{FV}(P) \setminus \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \overline{x}, \mathsf{result}, \overline{y}\\ \forall \overline{v}, \overline{w}.\ \vdash \{P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ c[\overline{v}/\overline{x}]\ \{\exists \overline{w'}.\ Q[\overline{v}/\overline{x}, \mathsf{res}/\mathsf{result}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}]\}\end{array}}{C \vdash \textbf{static}\ \tau\ m(\overline{\tau\ x})\ \textbf{req}\ P;\ \textbf{ens}\ Q;\ \{\ c\ \}\ \mathsf{ok}}$$

InstanceMethod
$$\frac{\begin{array}{c}\overline{y} = \mathsf{FV}(P) \setminus \mathsf{this}, \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \mathsf{this}, \overline{x}, \mathsf{result}, \overline{y}\\ \forall o, \overline{v}, \overline{w}.\ \vdash \{P[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ c[o/\mathsf{this}, \overline{v}/\overline{x}]\ \{\exists \overline{w'}.\ Q[o/\mathsf{this}, \overline{v}/\overline{x}, \mathsf{res}/\mathsf{result}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}]\}\end{array}}{C \vdash \textbf{instance}\ \tau\ m(\overline{\tau\ x})\ \textbf{req}\ P;\ \textbf{ens}\ Q;\ \{\ c\ \}\ \mathsf{ok}}$$

Implements
$$\frac{\begin{array}{c}\textbf{class}\ C(\cdots)\ \cdots\ \{\ \cdots\ \textbf{instance}\ \tau\ m(\overline{\tau\ x})\ \textbf{req}\ P';\ \textbf{ens}\ Q';\ \{\ c\ \}\ \cdots\ \}\\ \overline{y} = \mathsf{FV}(P) \setminus \mathsf{this}, \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \mathsf{this}, \overline{x}, \mathsf{result}, \overline{y}\\ \forall o, \overline{v}, \overline{w}.\ \vdash \{P[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ c[o/\mathsf{this}, \overline{v}/\overline{x}]\ \{\exists \overline{w'}.\ Q[o/\mathsf{this}, \overline{v}/\overline{x}, \mathsf{res}/\mathsf{result}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}]\}\end{array}}{\vdash C\ \mathsf{implements}\ \tau\ m(\overline{\tau\ x});\ \textbf{req}\ P;\ \textbf{ens}\ Q;}$$

Figure 7 Proof rules of the program logic for partial correctness.

An alternative approach would be to check *compatibility* of the class method's contract with the interface method's contract [11].

The rule of consequence uses validity of implications. We define $\models P \Rightarrow P'$ as $\forall h.\ I_{\text{fix}}, h \models P \Rightarrow I_{\text{fix}}, h \models P'$. In particular, we can fold and unfold predicates if we know the class of the object:

$$\frac{\textbf{class}\ C \cdots \ \{\ \cdots\ \textbf{predicate}\ p(\overline{\tau\ x}) = P;\ \cdots\ \}}{\models \mathsf{classOf}(o) = C \wedge o.p(\overline{v}) \Rightarrow P[o/\mathsf{this}, \overline{v}/\overline{x}] \qquad \models \mathsf{classOf}(o) = C \wedge P[o/\mathsf{this}, \overline{v}/\overline{x}] \Rightarrow o.p(\overline{v})}$$

We assume $\vdash$ *program* ok.

The proof rules are sound: $\vdash \{P\}\ c\ \{Q\} \Rightarrow \models \{P\}\ c\ \{Q\}$. By the following lemma:

▶ **Lemma 1.**

$$\begin{array}{l}\forall n, P, c, Q.\ \vdash \{P\}\ c\ \{Q\} \Rightarrow \\ \quad \forall h, h_0, h_F, \gamma.\ h = h_0 \uplus h_F \wedge I_{\text{fix}}, h_0 \models P \wedge (h, c) \Downarrow \gamma \Rightarrow \\ \quad\quad \gamma \neq \textbf{Failure}(n) \\ \quad\quad \wedge\ (\forall h', v.\ \gamma = (n, v, h') \Rightarrow \exists h'_0.\ h' = h'_0 \uplus h_F \wedge I_{\text{fix}}, h'_0 \models Q[v/\mathsf{res}])\end{array}$$

Proof. By well-founded induction on n. Fix some n_0 and assume the lemma holds for all $n < n_0$. We prove that it holds for $n = n_0$. By nested induction on the derivation of $\vdash \{P\}\ c\ \{Q\}$. ◀

3 Call Permissions

In the preceding section, we recalled a state-of-the-art approach from the literature for modular specification and verification of *partial correctness* properties of object-oriented programs. We are now ready to present the contributions of this paper. In this section, we extend the program logic of the preceding section to obtain a logic for *total correctness* properties. However, it is possible to write specifications in the logic of this section that overly constrain implementations, i.e. that distinguish implementations that are observationally indistinguishable. In the next section, we present a specification style for writing specifications in the logic of this section that perform proper information hiding.

How to extend the program logic of Sec. 2 so that it verifies the absence of infinite recursion? We wish to impose an additional proof obligation at method call sites, such that during any program execution only a finite number of method calls occur. Since we are already using separation logic, which can be interpreted as a logic of permissions, we introduce the notion of *call permissions*. If we make available to a program's main method only a finite stock of call permissions, and each call consumes a call permission, then it follows that an execution can perform only finitely many calls.

Note that we should not count call permissions merely using a natural number. This would mean each method's specification would state an upper bound on the number of calls it performs. That would seem to require much tedious and brittle bookkeeping, and cause problems if the number of calls depends on nondeterministic phenomena such as user input.

The well-known solution to the counting issue in termination proofs is the use of *well-founded relations*. A well-founded relation is one that admits no infinite descending chains, or, equivalently, where each nonempty set has a minimal element. In this paper, we will more specifically use *ordinals*, well-founded relations that are additionally strict total orders,

and for which useful conventional notations exist.[3][4]

We briefly review the ordinal theory used in this paper. The *finite ordinals* are the natural numbers, with their usual order. The set of finite ordinals is denoted ω. The *product* $\alpha \cdot \beta$ of two sets of ordinals is the set of pairs $(a, b) \in \alpha \times \beta$, with their *lexicographical ordering* (with the least significant element first): $(a, b) < (a', b')$ iff $b < b'$ or $b = b$ and $a < a'$. The *exponentiation* α^β of two sets of ordinals is the set of functions $f : \beta \to \alpha$ where only finitely many arguments map to nonzero values; the order is a generalization of the lexicographic order: $f < f'$ iff $f \neq f'$ and $f(b) < f'(b)$ where b is the maximum argument such that $f(b) \neq f'(b)$. In particular, ω^X yields the *multisets (or bags)* of elements of X, with *multiset order*. We denote bags using fat braces: $\{\!\!\{a, b, c\}\!\!\} = \mathbf{0} \uplus \{\!\!\{a\}\!\!\} \uplus \{\!\!\{b\}\!\!\} \uplus \{\!\!\{c\}\!\!\}$, where $\mathbf{0}$ denotes the empty multiset: $\mathbf{0} = \lambda x.\ 0$, and $M \uplus M'$ denotes multiset union: $M \uplus M' = (\lambda x.\ M(x) + M'(x))$. In order to descend down the multiset order starting from a multiset M, one can replace any element of M with any number of lesser elements of X, any number of times. For example, $\{\!\!\{0, 0, 1, 2, 2, 2\}\!\!\} < \{\!\!\{0, 0, 0, 3\}\!\!\}$.

Our program logic is based on the notion that at each point during a program's execution, it has a stock of call permissions in the form of a *bag of ordinals* $\Lambda \in \omega^{Ordinals}$ (for some fixed set of ordinals *Ordinals*). We admit ghost execution steps that reduce the stock of call permissions to a lesser one. Furthermore, at each call, an element is removed from the bag. It follows that the program terminates: an infinite execution would constitute an infinite descending chain in $\omega^{Ordinals}$.

We extend our separation logic with an assertion for call permissions:

$$P ::= o.f \mapsto v \mid P * P \mid \mathsf{call_perm}(\alpha) \mid \cdots$$

We interpret assertions under a predicate interpretation, a heap and a stock of call permissions:

$$\begin{array}{lcl} I, h, \Lambda \vDash \mathsf{call_perm}(\alpha) & \Leftrightarrow & \alpha \in \Lambda \\ I, h, \Lambda \vDash P * P' & \Leftrightarrow & \exists h_1, \Lambda_1, h_2, \Lambda_2.\ h = h_1 \uplus h_2 \wedge \Lambda = \Lambda_1 \uplus \Lambda_2 \\ & & \wedge\ I, h_1, \Lambda_1 \vDash P \wedge I, h_2, \Lambda_2 \vDash P' \end{array}$$

The meaning of Hoare triples is now defined as follows:

$$\vDash \{P\}\ c\ \{Q\} \quad \Leftrightarrow \quad \forall h, \Lambda, \gamma.\ I_{\mathrm{fix}}, h, \Lambda \vDash P \wedge (h, c) \Downarrow \gamma \Rightarrow \gamma \vDash_\Lambda Q$$

where satisfaction $\gamma \vDash_\Lambda Q$ of a postcondition by an outcome under a given stock of call permissions is now defined as follows:

$$\frac{\Lambda' \leq \Lambda \qquad I_{\mathrm{fix}}, h, \Lambda' \vDash Q[v/\mathsf{res}]}{(n, v, h) \vDash_\Lambda Q}$$

Notice that divergence is no longer considered to satisfy a postcondition.

The only proof rules that change are the rule of consequence and the rules for method calls. For the rule of consequence of our logic, we use a notion of implication that allows weakening of the stock of call permissions:

Conseq

$$\frac{P \sqsubseteq P' \qquad \vdash \{P'\}\ c\ \{Q'\} \qquad Q' \sqsubseteq Q}{\vdash \{P\}\ c\ \{Q\}}$$

[3] Our implementation of the proposed proof system (see Sec. 5) supports arbitrary well-founded relations.
[4] Technically, the ordinals are the equivalence classes of well-ordered sets under isomorphism. A well-ordered set is a set with a well-ordering, i.e. a well-founded strict total order. In an abuse of terminology, we will identify each such equivalence class with each of its members.

$$P \sqsubseteq P' \quad \Leftrightarrow \quad \forall h, \Lambda.\ I_{\text{fix}}, h, \Lambda \vDash P \Rightarrow \exists \Lambda' \leq \Lambda.\ I_{\text{fix}}, h, \Lambda' \vDash P'$$

We then have $\mathsf{call_perm}(1) \sqsubseteq \mathsf{call_perm}(0) * \mathsf{call_perm}(0)$, and, more generally, $\mathsf{call_perm}(1) \sqsubseteq \circledast_{i=1}^{n} \mathsf{call_perm}(0)$, for any n, where $\circledast_{i=a}^{b} P(i)$ represents iterated separating conjunction:

$$\circledast_{i=a}^{b} P(i) = \begin{cases} \mathsf{true} & \text{if } b < a \\ P(a) * \circledast_{i=a+1}^{b} & \text{otherwise} \end{cases}$$

In case i does not appear in P, we abbreviate $\circledast_{i=1}^{n} P$ to $n \cdot P$. So, for any $\alpha' < \alpha$ and any n, we have $\mathsf{call_perm}(\alpha) \sqsubseteq n \cdot \mathsf{call_perm}(\alpha')$.

The proof rules for method calls are as follows:

StaticCall

$$\frac{\begin{array}{c}\textbf{class}\ C \cdots \{ \cdots \textbf{static}\ \tau\ m(\overline{\tau\ x})\ \textbf{req}\ P;\ \textbf{ens}\ Q;\ \cdots \} \\ \overline{y} = \mathsf{FV}(P) \setminus \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \overline{x}, \mathsf{result}, \overline{y}\end{array}}{\vdash \{\mathsf{call_perm}(_) * P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ C.m(\overline{v})\ \{\exists \overline{w'}.\ Q[\overline{v}/\overline{x}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\}}$$

InstanceCall

$$\frac{\begin{array}{c}\textbf{interface}\ \iota\ \{ \cdots\ \tau\ m(\overline{\tau\ x});\ \textbf{req}\ P;\ \textbf{ens}\ Q;\ \cdots \} \\ \overline{y} = \mathsf{FV}(P) \setminus \mathsf{this}, \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \mathsf{this}, \overline{x}, \mathsf{result}, \overline{y} \qquad \theta = o/\mathsf{this}, \overline{v}/\overline{x}, \overline{w}/\overline{y}\end{array}}{\vdash \{\mathsf{call_perm}(_) * P[\theta]\}\ o.m(\overline{v})\ \{\exists \overline{w'}.\ Q[\theta, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\}}$$

Soundness follows from the following lemma:

▶ **Lemma 2.**

$$\begin{array}{l}\forall \Lambda, c, P, Q.\ \vdash \{P\}\ c\ \{Q\} \Rightarrow \\ \quad \forall h, h_0, h_F, \Lambda_0, \Lambda_F, \gamma.\ h = h_0 \uplus h_F \wedge \Lambda = \Lambda_0 \uplus \Lambda_F \wedge I_{\text{fix}}, h_0, \Lambda_0 \vDash P \wedge (h, c) \Downarrow \gamma \Rightarrow \\ \quad\quad \exists n, h', h'_0, v, \Lambda', \Lambda'_0.\ \gamma = (n, v, h') \wedge h' = h'_0 \uplus h_F \wedge \Lambda' = \Lambda'_0 \uplus \Lambda_F \wedge \Lambda' \leq \Lambda \\ \quad\quad\quad \wedge\ I_{\text{fix}}, h'_0, \Lambda'_0 \vDash Q[v/\mathsf{res}]\end{array}$$

Proof. By well-founded induction on $(|c|, \Lambda)$, where $|c|$ is the syntactic size of command c. By nested induction on the derivation of $\vdash \{P\}\ c\ \{Q\}$. ◀

4 Modular Specifications for Total Correctness

Now that we have call permissions, how do we use them to write modular specifications? Clearly, each method should require some call permissions from its caller in order to be able to perform calls itself. But how much? Which ordinal?

We introduce our modular specification approach incrementally, as follows. We first consider the case where the program performs *upcalls*, static method calls into lower[5] layers, only, in Sec. 4.1, and obtain a modular specification approach for this setting. In Sec. 4.2, we extend this approach so that it supports recursive static methods. Finally, in Sec. 4.3, we consider the general case, with dynamically bound instance method calls, and obtain the final version of the approach. Each of Sec. 4.1–4.3 corresponds to a different value for parameter *Ordinals* of the logic of Sec. 3. In Sec. 4.4, we show additional examples, illustrating how the approach deals with the advanced scenarios of interface methods taking objects as arguments and programs written in continuation-passing style.

[5] The clash of metaphors is unfortunate, but both terms are well-established.

4.1 Upcalls Only

It should not be necessary for a method to ask specifically for call permissions for each call it performs. In particular, as discussed in Sec. 1, we assume a model where a program consists of layers where each layer is built on top of lower layers to offer functionality to higher layers. In this model, a method's contract should not reveal whether, or how often, the method calls into lower layers. This is clearly an implementation detail that should not concern the method's clients. To support this notion, we assume that class definitions that appear earlier in a program text constitute lower layers with respect to class definitions that appear later. Similarly, within a class definition, we assume that methods that appear earlier constitute lower layers with respect to methods that appear later. To enable abstraction over calls to lower-layer methods, we will use class methods as ordinals, ordered by their position in the program text.

Using this approach, if all calls in a program call static methods in lower layers, it is sufficient for each method $C.m$ to require $\mathsf{call_perm}(C.m)$. Indeed, a valid proof outline is shown in Fig. 8.

Methods main and sqrt, before performing their nested call, using property $m' < m \Rightarrow \mathsf{call_perm}(m) \sqsubseteq 2 \cdot \mathsf{call_perm}(m')$, replace the incoming call permission qualified by their own name with two copies of a call permission qualified by their callee's name (using rule Conseq on p. 676). One copy is consumed at the start of the call, the other is passed into the callee as required by its precondition (per rule StaticCall on p. 677).

Formally, in this subsection we take

$$\begin{array}{rcl} \mathit{ClassMethods} & = & \{C.m \mid \textbf{class}\ C\ \cdots\ \{\ \cdots\ \tau\ m(\cdots)\ \cdots\}\} \\ \mathit{Ordinals} & = & \mathit{ClassMethods} \end{array}$$

4.2 Static Recursion

Of course, in most programs not all calls, even if they call static methods, call lower-layer methods, especially since our programming language does not have loops. How to deal with recursive methods? First of all, we assume that lower layers are developed before higher layers, and therefore lower layers never call static methods in higher layers. It follows that each cycle in a program's graph of calls to static methods is contained entirely within a single module. In our approach, then, all such members of a recursive cycle should be private to the module, and therefore their contracts need not be abstract. Separate public methods should be provided that call into the cycle but are not part of it.

An example is shown in Fig. 9.

Notice that each of the members of the cycle requires call permission for the maximum member. The contracts of isOddIter and isEvenIter are not abstract, but those of isOdd and isEven are.

Notice that the coefficient of the call permission in the contracts of the recursive methods serves the role of the classical recursion measure.

However, sometimes a recursion measure cannot easily be expressed as a simple natural number. To support such recursion measures, we move, for the set of ordinals that we use to qualify call permissions, from methods to pairs of *local ordinals* and methods (where the method is the most significant component), given some fixed set *LocOrd* of local ordinals. Formally, we take

$$\mathit{Ordinals} = \mathit{LocOrd} \cdot \mathit{ClassMethods}$$

```
class Math {
  static int sqrtHelper(int x)
    req 0 ≤ x ∧ call_perm(Math.sqrtHelper);
    ens true
  { ··· }
  static int sqrt(int x)
    req 0 ≤ x ∧ call_perm(Math.sqrt);
    ens true;
  {
    {2 · call_perm(Math.sqrtHelper)}
    Math.sqrtHelper(x)
  }
}
```

```
class Program {
  static void main()
    req call_perm(Program.main);
    ens true;
  {
    {2 · call_perm(Math.sqrt)}
    Math.sqrt(42)
  }
}
```

Figure 8 A program with calls to lower-layer static methods only.

```
class Math {
  static bool isOddIter(int x)
    req 0 ≤ x ∧
      x · call_perm(Math.isEvenIter);
    ens result = (x mod 2 = 1);
  {
    if x = 0 then false else
      Math.isEvenIter(x − 1)
  }
  static bool isEvenIter(int x)
    req 0 ≤ x ∧
      x · call_perm(Math.isEvenIter);
    ens result = (x mod 2 = 0);
  {
    if x = 0 then true else
      Math.isOddIter(x − 1)
  }
  static bool isOdd(int x)
    req 0 ≤ x ∧ call_perm(Math.isOdd);
    ens result = (x mod 2 = 1);
  { Math.isOddIter(x) }
  static bool isEven(int x)
    req 0 ≤ x ∧ call_perm(Math.isEven);
    ens result = (x mod 2 = 0);
  { Math.isEvenIter(x) }
}
```

Figure 9 Recursion measured by a natural number.

Public methods $C.m$ should request $\mathsf{call_perm}((0, C.m))$. A classic example is the Ackermann function:

```
class Math {
  static int ackermannIter(int m, int n)
    req 0 ≤ m ∧ 0 ≤ n ∧ call_perm(((m, n), Math.ackermannIter));
    ens result = Ack(m, n);
  {
    if n = 0 then m + 1
    else if m = 0 then Math.ackermannIter(1, n − 1)
    else {
      int r := Math.ackermannIter(m − 1, n);
      Math.ackermannIter(r, n − 1)
    }
  }
  static int ackermann(int m, int n)
    req 0 ≤ m ∧ 0 ≤ n ∧ call_perm((0, Math.ackermann));
    ens result = Ack(m, n);
  { Math.ackermannIter(m, n) }
}
```

The proof of method ackermann uses the property $\forall m, n.\ ((m, n), \mathsf{Math.ackermannIter}) < (0, \mathsf{Math.ackermann})$. This example assumes $\omega \cdot \omega \subseteq \mathit{LocOrd}$.

4.3 Dynamic Binding

We are now ready to consider the case of programs that call interface methods. Consider a method integrate for computing the integral of a real function over an interval:

```
interface RealFunc {
  double apply(double x);
}
class Math {
  static double integrate(double a, double b, RealFunc f)
  { · · · }
}
```

What call permissions should method integrate request of its caller? Clearly, the method should be allowed to call method apply of object f. And it should be allowed to call it not just once, but arbitrarily often. Since the calls of f.apply might occur inside recursive helper functions measured by arbitrary ordinals, there is no single ordinal that can obviously serve as an upper bound. Furthermore, method integrate should be allowed to pass f to library methods in lower layers, and those should themselves be specified abstractly without revealing how often they call f.

To solve this problem, we move, for the set of ordinals that we use to qualify call permissions, from pairs of local ordinals and methods to pairs of local ordinals and *bags of methods*:

$$\begin{aligned} \mathit{MethodBags} &= \omega^{\mathit{ClassMethods}} \\ \mathit{Ordinals} &= \mathit{LocOrd} \cdot \mathit{MethodBags} \end{aligned}$$

Note that set *ClassMethods* includes both the static methods and the instance methods.

Public methods $C.m$ in programs that do not call interface methods should request call_perm($(0, \{\!\{C.m\}\!\})$).

The following contract for method integrate allows it to call f.apply arbitrarily often, and to delegate a similar permission to lower-layer static methods:

static double integrate(**double** a, **double** b, RealFunc f)
req call_perm($(0, \{\!\{$Math.integrate, f.apply$\}\!\})$);

Note that we use $o.m$ as a shorthand for classOf(o).m.

However, we are not done. Indeed, when calling f.apply, we need not just the call permission that is consumed by the call itself, but also the call permissions requested by f.apply's precondition. What should those be?

f.apply should be allowed to call static methods at layers below itself, so it should at least receive a call permission qualified by its own name. However, there are other methods that it should be allowed to call as well. Indeed, through the fields of its this object, i.e. through the fields of f, this method may be able to reach directly or indirectly any number of objects and might need to perform any number of calls on any number of methods thereof. The method should request permission for those calls as well.

But how can it abstractly request permission to call these methods, hidden inside its private data structures, with whose existence its clients should not otherwise be concerned?

We solve this problem by allowing the bag of methods reachable from an object to be named abstractly by exposing it as an argument of the predicate family that describes the object.

Consider first the partial-correctness specification for interface RealFunc in Fig. 10(a). We extend it for total correctness as shown in Fig. 10(b).

We adapt the contract of method integrate accordingly:

static double integrate(**double** a, **double** b, RealFunc f)
req f.RealFunc(d) $*$ call_perm($(0, \{\!\{$Math.integrate$\}\!\} \uplus$ d$)$);

A simple implementation of interface RealFunc is shown in Fig. 11.

Notice that method createLinearFunc's postcondition provides an upper bound on d. This enables the caller (who is necessarily in a higher layer than createLinearFunc) to produce the call permissions required to call result.apply. Notice also that this upper bound does not constrain method createLinearFunc's implementation, if we assume that a method only allocates (through **new**) objects of classes defined in its own layer or in lower layers.

A slightly more involved implementation is shown in Fig. 12.

Notice that class Sum's instance of predicate family RealFunc defines its d parameter (which we call the *dynamic depth* since it gives a measure of the number of layers of abstraction of which the object is composed) as the multiset union of its own apply method and the referenced objects' dynamic depths. Indeed, as a general pattern, an object's dynamic depth should typically be defined as the union of its own methods and the dynamic depths of the objects stored in its fields. This allows the object to call those objects' methods, assuming their contracts follow the standard pattern exemplified by the contract of RealFunc.apply.

Notice also how this definition enables the successful verification of method apply.

Method createSum's precondition follows the general pattern: request a call permission qualified by a multiset that is the union of the method itself and the dynamic depths of any objects being passed into the method. Its postcondition follows a general pattern for methods that return a new object: the new object's dynamic depth is bounded by the same

```
interface RealFunc {
  predicate RealFunc();
  double apply(double x);
    req this.RealFunc();
    ens this.RealFunc();
}
```

(a) Partial correctness

```
interface RealFunc {
  predicate RealFunc(MethodBag d);
  double apply(double x);
    req this.RealFunc(d) ∗ call_perm((0, d));
    ens this.RealFunc(d);
}
```

(b) Total correctness

Figure 10 Annotations for interface RealFunc.

```
class LinearFunc(double a, double b) implements RealFunc {
  predicate RealFunc(MethodBag d) = this.a ↦ _ ∗ this.b ↦ _ ∧ d = {[this.apply]};
  double apply(double x) { double a := this.a; double b := this.b; a · x + b }
  static RealFunc createLinearFunc(double a, double b)
    req call_perm((0, {[LinearFunc.createLinearFunc]}));
    ens result.RealFunc(d) ∧ d < {[LinearFunc.createLinearFunc]};
  { new LinearFunc(a, b) }
}
```

Figure 11 A simple implementation of interface RealFunc.

```
class Sum(RealFunc f1, RealFunc f2) implements RealFunc {
  predicate RealFunc(MethodBag d) =
    this.f1 ↦ f1 ∗ f1.RealFunc(d1) ∗ this.f2 ↦ f2 ∗ f2.RealFunc(d2)
    ∧ d = {[this.apply]} ⊎ d1 ⊎ d2;
  double apply(double x) {
    RealFunc f1 := this.f1; RealFunc f2 := this.f2;
    double r1 := f1.apply(x); double r2 := f2.apply(x); r1 + r2
  }
  static RealFunc createSum(RealFunc f1, RealFunc f2)
    req f1.RealFunc(d1) ∗ f2.RealFunc(d2) ∗ call_perm((0, {[Sum.createSum]} ⊎ d1 ⊎ d2));
    ens result.RealFunc(d) ∧ d < {[Sum.createSum]} ⊎ d1 ⊎ d2;
  { new Sum(f1, f2) }
}
```

Figure 12 An implementation of interface RealFunc.

```
class Math {
  static double integrateIter(double a, double dx, int n, RealFunc f)
      req 0 ≤ n ∧ f.RealFunc(d) ∗ call_perm((n, {[Math.integrateIter]} ⊎ d));
      ens f.RealFunc(d);
  {
    if n = 0 then 0 else {
      double y := f.apply(a); double ys := Math.integrateIter(a + dx, dx, n − 1, f);
      y × dx + ys
    }
  }
  static double integrate(double a, double b, RealFunc f)
  { Math.integrateIter(a, (b − a)/1000, 1000, f) }
}
class Program {
  static main()
      req call_perm((0, {[Program.main]}));
      ens true;
  {
    {12 · call_perm((0, {[2 · LinearFunc, 2 · Sum, Math]}))}
    RealFunc f1 := LinearFunc.createLinearFunc(2, 3);
    RealFunc f2 := LinearFunc.createLinearFunc(5, 6);
    RealFunc f3 := LinearFunc.createLinearFunc(5, 6);
    RealFunc f4 := Sum.createSum(f1, f2);
    RealFunc f5 := Sum.createSum(f3, f4);
    Math.integrate(0, 100, f5)
  }
}
```

Figure 13 An implementation of method integrate and a client program.

multiset used to qualify the call permission in the precondition. It follows that any caller of this method can also call the new object's methods.

An implementation of method integrate and an example client program are shown in Fig. 13.

The proof outline for method main starts by reducing the incoming call permission to twelve copies of a call permission that is greater than each of the call permissions required for the six calls and the six preconditions. (We use a class name as an abbreviation for its greatest method.) Indeed, we have the inequalities shown in Fig. 14.

4.4 Further Examples

An example of an interface method that takes as an argument another object is shown in Fig. 15. Notice that method intersects' precondition asserts a single call permission qualified by the multiset union of the dynamic depth of the receiver and the dynamic depth of the argument object.

Our approach supports methods written in continuation-passing style (CPS). To illustrate

$$\begin{array}{ll}
\mathsf{d1}, \mathsf{d2}, \mathsf{d3} & < \{\!\{\mathsf{LinearFunc.createLinearFunc}\}\!\} \\
\mathsf{d4} & < \{\!\{\mathsf{Sum.createSum}\}\!\} \uplus \mathsf{d1} \uplus \mathsf{d2} \\
 & < \{\!\{\mathsf{Sum.createSum}, \mathsf{LinearFunc.createLinearFunc}\}\!\} \\
\mathsf{d5} & < \{\!\{\mathsf{Sum.createSum}\}\!\} \uplus \mathsf{d3} \uplus \mathsf{d4} \\
 & < \{\!\{2 \cdot \mathsf{LinearFunc.createLinearFunc}, 2 \cdot \mathsf{Sum.createSum}\}\!\}
\end{array}$$

Figure 14 Inequalities relevant for the proof of the integrate client program. Symbols $\mathsf{d1}, \ldots, \mathsf{d5}$ denote the dynamic depths of objects $\mathsf{f1}, \ldots, \mathsf{f5}$.

```
interface IntSet {
  predicate IntSet(MethodBag d);
  bool contains(int x);
    req this.IntSet(d) ∗ call_perm((0, d));
    ens this.IntSet(d);
  bool intersects(IntSet other);
    req this.IntSet(d) ∗ other.IntSet(do) ∗ call_perm((0, d ⊎ do));
    ens this.IntSet(d) ∗ other.IntSet(do);
}
class Empty() implements IntSet {
  predicate IntSet(MethodBag d) = d = {{this.∗}};
  bool contains(int x) { false }
  bool intersects(IntSet other) { false }
  static IntSet createEmpty()
    req call_perm((0, {{Empty.createEmpty}}));
    ens result.IntSet(d) ∧ d < {{Empty.createEmpty}};
  { new Empty() }
}
class Insert(int elem, IntSet set) implements IntSet {
  predicate IntSet(MethodBag d) =
    this.elem ↦ elem ∗ this.set ↦ set ∗ set.IntSet(ds) ∧ d = {{this.∗}} ⊎ ds;
  bool contains(int x) {
    int elem := this.elem;
    if x = elem then true else { IntSet set := this.set; set.contains(x) }
  }
  bool intersects(IntSet other) {
    int elem := this.elem; bool contains := other.contains(elem);
    if contains then true else { IntSet set := this.set; set.intersects(other) }
  }
  static IntSet createInsert(int elem, IntSet set)
    req set.IntSet(ds) ∗ call_perm((0, {{Insert.createInsert}} ⊎ ds));
    ens result.IntSet(d) ∧ d < {{Insert.createInsert}} ⊎ ds;
  { new Insert(elem, set) }
}
```

Figure 15 An interface method that takes as an argument another object. $\{\!\{o.*\}\!\}$ denotes the bag of all instance methods of o.

```
interface ContainsCont {
  predicate ContainsCont(IntSet set, MethodBag d);
  Nothing invoke(bool result);
    req this.ContainsCont(set, d) ∗ set.IntSet(d);
}
interface IntersectsCont {
  predicate IntersectsCont(IntSet set, MethodBag d, IntSet other, MethodBag do);
  Nothing invoke(bool result);
    req this.IntersectsCont(set, d, other, do) ∗ set.IntSet(d) ∗ other.IntSet(do);
}
interface IntSet {
  Nothing containsCPS(int x, ContainsCont cont);
    req this.IntSet(d) ∗ call_perm((0, d)) ∗ cont.ContainsCont(this, d);
  Nothing intersectsCPS(IntSet other, IntersectsCont cont);
    req this.IntSet(d) ∗ other.IntSet(do) ∗ call_perm((0, d ⊎ do))
      ∗ cont.ContainsCont(this, d, other, do);
}
```

Figure 16 Continuation-passing-style versions of contains and intersects. All postconditions are false and are not shown.

this, we add CPS versions of methods contains and intersects to the IntSet example. See Figs. 16 and 17. After methods containsCPS and intersectsCPS are finished computing their result value, they do not return to the caller (as indicated by return type Nothing, which has no values); rather, they call method invoke of the *continuation* object cont, passing the result value as an argument. The predicate definitions and constructor methods remain unchanged and are not repeated.

Notice that the continuation interfaces do not follow the general specification pattern. Indeed, since the invoke methods are invoked only once, any call permissions they need can be passed via the ContainsCont or IntersectsCont predicate, respectively.

Notice that InvokeCont1.invoke and InvokeCont2.invoke each require a single call permission in order to perform the nested invoke call. It is passed via the predicate. The ordinal is irrelevant and is existentially quantified. In contrast, InvokeCont3.invoke needs to call set.intersectsCPS and for that needs a properly qualified call permission. The call permission that the InvokeCont3 object needs to pass to the InvokeCont2 object can be derived from it.

5 Implementation

We integrated the logic into the program verification tool VeriFast. Although VeriFast supports C and Java, for now we have added support for verification of termination only for C programs. We introduced the function specification clause **terminates**, to indicate that a function should terminate. In order to reduce specification overhead for functions that do not perform callbacks, our implementation offers a ghost command that allows a function to produce out of thin air any call permission whose bag of functions is less than

```
class Empty() implements IntSet {
  Nothing containsCPS(int x, ContainsCont cont) { cont.invoke(false) }
  Nothing intersectsCPS(IntSet other, IntersectsCont cont) { cont.invoke(false) }
}
class InsertCont1(ContainsCont cont) implements ContainsCont {
  predicate ContainsCont(IntSet set, MethodBag d) =
    this.cont ↦ cont ∗ cont.ContainsCont(set0, d0)
    ∗ set0.elem ↦ _ ∗ set0.set ↦ set ∗ call_perm(_) ∧ d0 = {[set0.∗]} ⊎ d;
  Nothing invoke(bool result) { ContainsCont cont := this.cont; cont.invoke(result) }
}
class InsertCont2(IntersectsCont cont) implements IntersectsCont {
  predicate IntersectsCont(IntSet set, MethodBag d, IntSet other, MethodBag do) =
    this.cont ↦ cont ∗ cont.IntersectsCont(set0, d0, other, do)
    ∗ set0.elem ↦ _ ∗ set0.set ↦ set ∗ call_perm(_) ∧ d0 = {[set0.∗]} ⊎ d;
  Nothing invoke(bool result) { IntersectsCont cont := this.cont; cont.invoke(result) }
}
class InsertCont3(Insert set0, IntSet other, IntersectsCont cont) implements ContainsCont {
  predicate ContainsCont(IntSet set, MethodBag d) =
    this.other ↦ set ∗ this.cont ↦ cont ∗ cont.IntersectsCont(set0, ds0, set, d)
    ∗ this.set0 ↦ set0 ∗ (set0.IntSet(ds0) ∧ ds0 = {[set0.∗]} ⊎ ds1)
    ∗ call_perm((0, {[InsertCont3.invoke]} ⊎ ds1 ⊎ d));
  Nothing invoke(bool result) {
    Insert set0 := this.set0; IntSet other := this.other; IntersectsCont cont := this.cont;
    if result then cont.invoke(true) else {
      IntSet set := set0.set; IntersectsCont cont1 := new InsertCont2(cont);
      set.intersectsCPS(other, cont1)
    }
  }
}
class Insert(int elem, IntSet set) implements IntSet {
  Nothing containsCPS(int x, ContainsCont cont) {
    int elem := this.elem; if x = elem then cont.invoke(true) else {
      IntSet set := this.set; ContainsCont cont1 := new InsertCont1(cont);
      set.containsCPS(x, cont1)
    }
  }
  Nothing intersectsCPS(IntSet other, IntersectsCont cont) {
    int elem := this.elem; ContainsCont cont1 := new InsertCont3(this, other, cont);
    other.containsCPS(elem, cont1)
  }
}
```

Figure 17 Implementations of containsCPS and intersectsCPS in classes Empty and Insert.

itself (considered as a singleton bag). In exchange, our implementation consumes at a call site not just any call permission, but only a call permission whose function bag includes the function being called:

$$\frac{\begin{array}{c} f \in d \\ \textbf{function } f(\overline{x}) \textbf{ req } P;\ \textbf{ens } Q;\ \{\ c\ \} \qquad \overline{y} = \mathsf{FV}(P) \setminus \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \overline{x}, \mathsf{result}, \overline{y} \end{array}}{f_0 \vdash \{\mathsf{call_perm}((\alpha, d)) * P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ f(\overline{v})\ \{\exists \overline{w'}.\ Q[\overline{v}/\overline{x}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\}}$$

$$\frac{\begin{array}{c} \forall \overline{v}, \overline{w}.\ f \vdash \{P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ c\ \{\exists \overline{w'}.\ Q[\overline{v}/\overline{x}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\} \\ \overline{y} = \mathsf{FV}(P) \setminus \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \overline{x}, \mathsf{result}, \overline{y} \end{array}}{\vdash \textbf{function } f(\overline{x}) \textbf{ req } P;\ \textbf{ens } Q;\ \{\ c\ \}\ \mathsf{ok}}$$

$$\frac{d < \{\!\!\{f\}\!\!\}}{f \vdash \{\mathsf{true}\}\ \textbf{produce_call_perm}\ \{\mathsf{call_perm}((\alpha, d))\}}$$

Our implementation is included in the latest VeriFast release, which is available at `http://www.cs.kuleuven.be/~bartj/verifast/`.

The distribution includes, in the directory examples/termination, the examples simple_recursion.c (the isEven example), ackermann.c, funcptr.c (corresponding to the IntFunc example of this paper), and cons.c (corresponding to the IntSet example).

6 Related Work

We are not aware of existing approaches for modular specification and verification of termination of object-oriented programs. However, work on modular verification of termination in different settings does exist.

The proof assistant Coq includes a pure functional programming language with higher-order functions. Coq checks that all functions terminate. However, Coq's type system prevents a function from being passed as an argument to itself. Our approach supports methods that call themselves through dynamic binding, and can prove their termination.

Koka [4] is a functional programming language with effect inference, including the divergence effect. However, the inference algorithm is limited: it rules out recursion through the heap, which our approach supports.

Dafny [5] is a programming language that supports verification of termination, with powerful metrics. However, Dafny does not support dynamic binding of method calls.

Most closely related to ours is the work, e.g. [1, 12, 6], on proving well-definedness of specifications for object-oriented programs where the specifications themselves involve calls of methods of the program being specified. In most such approaches, in order to ensure that such specifications make sense and that axioms generated from such specifications are consistent, proof obligations are imposed to verify that methods called from specifications are *pure* (i.e., side-effect-free) and that they *terminate*. Our notion of *dynamic depth* of a data structure can be seen as a refinement of the *depth of the ownership tree* (a natural number) used as a recursion measure by some of this work [1, 6]. In these approaches, the ownership graph is frozen for the duration of the execution of a pure method, so if calls descend down an ownership tree, they terminate. In our approach, however, to support non-pure methods that create new data structures composed of lower-layer classes, we track not just the number of objects comprising a data structure; rather, we track the multiset of the modules that define their classes.

7 Conclusion

We propose an approach for the modular specification and verification of total correctness properties of sequential object-oriented programs involving dynamically bound method calls. As far as we know, it is the first such approach. We propose a specification style that does not constrain implementations unnecessarily.

We have implemented our approach in a verification tool and validated it on a handful of small but challenging example programs. Further experimentation is needed, however, to see if our approach conveniently handles all program patterns.

In the extended version of this paper [3], we discuss how to combine our approach with an approach for modular verification of absence of deadlock, such as [7], to obtain an approach for modular verification of termination of multithreaded programs. By using dynamic depths as wait levels, the approach allows acquisitions of private locks to be introduced into existing methods without changing their contracts. We also show how the approach supports proving termination of compare-and-swap loops, and proving liveness of non-terminating programs.

Acknowledgements. The authors would like to thank Matthew Parkinson for his helpful comments, and K. Rustan M. Leino for pointing out to them the usefulness of ordinals and multiset order for termination verification. This work was supported in part by EU project ADVENT and by project G.0058.13 of the Research Foundation – Flanders (FWO).

References

1. Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, 2006.
2. Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.
3. Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. Modular termination verification: extended version. Technical Report CW 680, Dept. Comp. Sci., KU Leuven, 2015.
4. Daan Leijen. Koka: Programming with row polymorphic effect types. In *Mathematically Structured Functional Programming*, 2014.
5. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.
6. K. Rustan M. Leino and Ronald Middelkoop. Proving consistency of pure methods and model fields. In *FASE*, 2009.
7. K. Rustan M. Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In *ESOP*, 2010.
8. Keiko Nakata and Tarmo Uustalu. Trace-based coinductive operational semantics for While. In *TPHOLs*, 2009.
9. Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.
10. Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *POPL*, 2005.
11. Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *POPL*, 2008.
12. Arsenii Rudich, Ádám Darvas, and Peter Müller. Checking well-formedness of pure-method specifications. In *FM*, 2008.

Framework for Static Analysis of PHP Applications*

David Hauzar and Jan Kofroň

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic

Abstract

Dynamic languages, such as PHP and JavaScript, are widespread and heavily used. They provide dynamic features such as dynamic type system, virtual and dynamic method calls, dynamic includes, and built-in dynamic data structures. This makes it hard to create static analyses, e.g., for automatic error discovery. Yet exploiting errors in such programs, especially in web applications, can have significant impacts. In this paper, we present static analysis framework for PHP, automatically resolving features common to dynamic languages and thus reducing the complexity of defining new static analyses. In particular, the framework enables defining value and heap analyses for dynamic languages independently and composing them automatically and soundly. We used the framework to implement static taint analysis for finding security vulnerabilities. The analysis has revealed previously unknown security problems in real application. Comparing to existing state-of-the-art analysis tools for PHP, it has found more real problems with a lower false-positive rate.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Static analysis, abstract interpretation, dynamic languages, PHP, security

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.689

Supplementary Material ECOOP Artifact Evaluation approved artifact available at
`http://dx.doi.org/10.4230/DARTS.1.1.11`

1 Introduction

To analyze programs precisely and soundly, static analysis needs to resolve method calls, include statements, and accesses to data structures. Since in dynamic languages, targets of method calls and include statements can depend on information about values (and types) of expressions, value analysis tracking values of all primitive data types present in the language needs to be performed. Moreover, due to frequent use of dynamic data structures such as associative arrays and objects, value analysis needs to be combined with heap analysis. These depend on each other also the other way round – since array indices and object properties can be accessed using arbitrary expressions, heap analysis needs value analysis to evaluate these expressions. This makes any end-user static analysis that takes this into account overly complex.

* This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S and by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, http://www.spark-2014.org/proofinuse) of the French national research organization.

© David Hauzar and Jan Kofroň;
licensed under Creative Commons License CC-BY

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 689–711

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 Propagation of tainted data.

Lattice	$(L, \sqsubseteq, \sqcup)$	$(Bool, \implies, \vee)$	
Initial value	$init(v)$	$true$	if $v \in \$_\texttt{POST} \cup \$_\texttt{GET} \cup \ldots$
		$false$	otherwise
Transfer function	$[\![LHS = RHS]\!]$	$var = \bigvee_{r \in RHS} r$	if $var \in LHS$
		$var = var$	otherwise
	$[\![st]\!]$	$var = var$	if st is not assignment

In this paper we present a static analysis framework for languages with dynamic features [10] based on abstract interpretation [1]. The framework automatically resolves dynamic features and makes it possible to define static analyses without taking these features explicitly into account.

In particular, our contributions include:

- The architecture of the static analysis framework for dynamic languages and the way dynamic features are automatically resolved.
- Description of all necessary analyses that are needed to automatically resolve dynamic features. We define value analysis that tracks values of all primitive data types of PHP. We articulate our assumptions on heap analysis to take dynamic index and property accesses into account – indices and properties are created when they are accessed for the first time and accesses can be performed using arbitrary value expressions, yielding statically unknown values.
- Composition of all necessary analyses allowing to define these analyses independently. Here the main challenge is defining the interplay of value analysis and heap analysis taking dynamic features into account. The composition is sound; if the analyses being composed are sound, the resulting analysis is sound as well.

2 Motivation

As a motivation example, consider static taint analysis, which is often used for security analysis of web applications. It can be used for detection of security problems, e.g., vulnerability of an application to SQL injection and cross-site scripting attacks. Static taint analysis can be described as follows. The program point that reads user-input, session ids, cookies, or any other data that can be manipulated by a potential attacker is called *source*, while a program point that prints out data to a browser, queries a database, etc. is referred to as *sink*. Data at a given program point are *tainted* if they can pass from a source to this program point. Tainted data are *sanitized* if they are processed by a sanitization routine (e.g., `htmlspecialchars` in PHP) to remove potentially malicious parts. Program is *vulnerable* if it contains a sink that uses data that are tainted and not sanitized.

Static taint analysis can be performed by computing the propagation of tainted data and then checking whether tainted data can reach a sink. The specification of forward data-flow analysis computing the propagation of tainted data is shown in Tab. 1[1]. The analysis is specified by the lattice of data-flow facts and lattice operators, the initial values of variables, and the transfer function.

Consider now the code in Fig. 1. The code contains two vulnerabilities to XSS attack [7]. The first vulnerability corresponds to the call at line (25), the second vulnerability corresponds

[1] For simplicity we omit the specification of sanitization.

```
1  class Templ {
2    function log($msg) {...}
3  }
4  class Templ1 : Templ {
5    function show($msg) { sink($msg); }
6  }
7  class Templ2 : Templ {
8    function show($msg) { not_sink($msg); }
9  }
10 function initialize(&$users) {
11   $users['admin']['addr'] = get_admin_addr_from_db();
12 }
13 switch (DEBUG) {
14   case true: $mode = "log"; break;
15   default: $mode = "show";
16 }
17 switch ($_GET['skin']) {
18   case 'skin1': $t = new Templ1(); break;
19   default: $t = new Templ2();
20 }
21 initialize($users);
22 $id = $_GET['userId'];
23 $users[$id]['name'] = $_GET['name'];
24 $users[$id]['addr'] = $_GET['addr'];
25 $t−>$mode($users[$id]['name']);
26 $t−>$mode($users['admin']['addr']);
```

Figure 1 Running example.

to the call at line (26). In both cases the method `show` of `Templ1` can be called (line (5)) with the parameter `$msg` being tainted and going to the sink. Taint analysis defined using our framework uses just the information in Tab. 1 and can still detect both vulnerabilities. This is possible only because the framework automatically resolves control flow and accesses to built-in data structures. That is, the framework computes that the variable `$t` can point to objects of types `Templ1` and `Templ2` and that the variable `$mode` can contain values `"show"` and `"log"`. Based on this information, it automatically resolves calls at lines (25) and (26). As the framework automatically reads the data from and updates the data to associative arrays and objects, tainted data written at line (23) are read at line (25). Moreover, at line (24), the tainted data are automatically propagated to index `$users['admin']['addr']` defined at line (11). Consequently, the access of this index at line (26) reads this tainted data.

3 Static Analysis Framework

The architecture of the framework is shown in Fig. 2. The analysis is split into two phases. In the first phase, the framework computes control flow of the analyzed program together with the shape of the heap and information about values of variables, array indices and object properties and evaluates expressions used for accessing data. The control flow is captured in the intermediate representation (IR), while the other information can be accessed using the data representation. In the second phase, end-user analyses of the constructed IR are performed.

Data representation allows accessing analysis states. In particular, it allows reading and writing values, and modifying shape of data structures. Next, it performs join and widening of analysis states and defines their partial order. Importantly, data representation defines

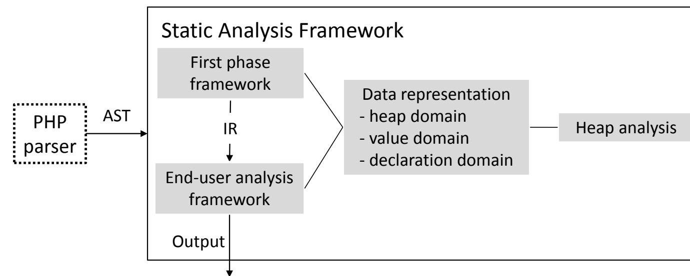


Figure 2 Architecture of the framework.

the interplay of heap, value, and declaration analyses allowing each analysis to define these operations independently on other analyses.

The implementation of heap analysis tracks the shape of the heap and must provide information that the value analysis needs to read values from data structures, update values to data structures, and to join values stored in data structures.

The implementation of the first phase must provide information necessary for computing control flow of the program and the information that the heap analysis needs to access data. That is, it must define value analysis that tracks values of PHP primitive types, evaluates value expressions modeling native operators, native functions, and implicit conversions. Next, the implementation must define declaration analysis handling declarations of functions, classes, and constants. Finally, it must compute targets of throw statements, include statements, and function and method calls.

The implementations of end-user analyses define additional value analyses. In contrast to value analysis for the first phase, which must track values of PHP primitive types, end-user value analyses can use an arbitrary value domain. This is possible because

1. control flow is already computed,
2. the shape of the heap is computed and dynamic data accesses are resolved (i.e., value expressions specifying data accesses are evaluated). That is, all information that the data representation needs to determine accessed variables, array indices, and object properties is available.
3. Data representation combines heap, value, and declaration analyses automatically.

3.1 Intermediate Representation

The intermediate representation (IR) of our analysis is a graph, in which each node contains an instruction. There are two types of nodes in the graph – *value nodes* and *non-value nodes*. Value nodes compute and store representation of values while non-value nodes perform other actions. The graph has two types of edges. *Flow edges* represent potential control flow between instructions of the program – they define ordering in which program instructions can be executed. *Value edges* connect nodes that use values (e.g., operators) with nodes that represent these values (e.g., operands).

Each node has associated an analysis state stored in data representation. The state is modified by transfer function defined for the node and the resulting state is propagated to successor nodes connected with flow edges. If a node has more predecessors the states of predecessors are joined.

Note that transfer functions for most of the value nodes are defined as identity – they do not modify the analysis state. That is, most of the value nodes just compute values (e.g., evaluate expressions) or compute information that specify data access to values (e.g.,

compute possible names of variables that they represent). This information is stored in data representation, but it is not a part of the analysis state and thus it is not propagated to successor nodes. Instead, nodes that use these values (e.g. operator nodes) are connected with value nodes (e.g. operands) using value edges. If an operand value is needed when evaluating the operator, the value edge is used to get the value from the operand.

▶ **Example 1.** As an example, consider the intermediate representation corresponding to the statement `$$a=b($c)`. The statement assigns the value computed by function `b` to a variable with the name given by the value of variable `$a`. The resulting intermediate representation is depicted in Fig. 3. Note that the node corresponding to the assignment instruction is connected using a value edge with the source of the assignment (the node containing the value computed by the function `b`) and with the target of the assignment (the node representing the assigned variable – `$`). Next, the latter node is connected using a value edge with the node representing possible names of the assigned variable (node `$a`).

The nodes can be of different types. In the following, we denote value nodes by adding superscript V; for each node, its parameters are the value nodes that are connected with the node using value edges:

variable$^V[n^V]$**:** represents a variable – stores the information necessary for accessing the variable in data representation. The parameter n^V is the value node that represents the name of the variable. Note that reading n^V yields an arbitrary value from the abstract string domain and can thus represent more concrete string values – names. Consequently, the variable node can represent more concrete variables.

property-use$^V[o^V, f^V]$**, index-use**$^V[a^V, i^V]$**:** property-use^V stores the information for accessing a property of the given object. Parameter o^V is the value node storing the representation of the object and f^V is the value node storing the name of the property. Again, reading o^V and f^V yields abstract values and the property-use^V node can get representation of more properties. The index-use^V is similar and it is used for accessing arrays.

assign$^V[l^V, r^V]$**:** represents the assignment of the right operand r^V to the left operand l^V and stores the information for accessing this value. While the parameter l^V is a value node whose type can be variable, property-use, and item-use, the parameter r^V is an arbitrary value node.

alias$^V[l^V, r^V]$**:** represents the alias statement. The alias statement is similar to the assignment statement. However, besides performing the assignment, the alias statement creates explicit alias between its parameters and both parameters of the alias statement must be variables, object properties, or array indices.

expression$^V[e, o_1^V, ..., o_n^V]$**:** represents the expression e with operands $o_1^V, ..., o_n^V$. It stores the representation of the result.

assume$[c]$**:** represents assumption implied, e.g., by *if* and *while* statements. It indicates whether the condition c is satisfiable. If the condition is unsatisfiable, the flow is not propagated to descendant nodes. If the condition is satisfiable, the analysis state is refined according to the condition and then propagated to descendant nodes.

constant-declaration$[d]$**:** represents declaration of a constant.

function-declaration$[d]$**:** represents declaration of a function.

class-declaration$[d]$**:** represents declaration of a class.

call$^V[n^V, o^V, a]$**, construct**$^V[n^V, a]$**:** represents a call of a function whose name is specified using the value node n^V on an object specified using the value node o^V with arguments specified using a list of value nodes a. The construct^V nodes are similar to call^V nodes

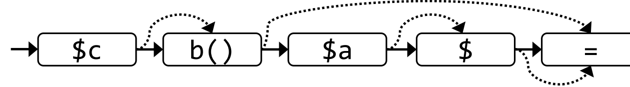


Figure 3 Intermediate representation of the statement `$$a=b($c)`. Solid edges are flow edges, dashed edges are value edges.

and are used for `new` expressions. Note that reading n^V, o^V, and elements of a yields abstract values that can represent more concrete values.

return$[e^V]$**:** represents a return from a function. e^V represents the value of a return expression.

include$[p^V]$**:** represents the inclusion of the script given by the path specified by the value node p^V. Again, a path can represent more concrete values.

eval$[c^V]$**:** evaluates the code fragment specified by value node c^V.

native-method^a[]**:** represents execution of a native method or a native function with arguments specified using a list of value nodes a.

extension$[f, a]$**:** is used to dynamically extend the control from IR node f. This is necessary when the information needed to determine the control flow from the node is computed by the analysis. This is the case of calls to functions, methods and constructors, and the include and eval statements. During analysis, for each dynamically discovered control flow from the node, a single extension node is added. Parameter f is the node that is extended. Parameter a is used in the cases that the control flow is extended because of a function, method, and constructor call and it is a list of value nodes representing parameters of the call.

extension-sink$[n]$**:** represents a join point of all the extensions of node n.

try-scope-start$[c]$ **and try-scope-end**$[c]$ **:** represent the start and the end of a `try` block. Parameter c represents catch blocks associated with the `try` block.

throw$[v^V]$**:** represents the `throw` statement. Parameter v^V is the node representing the value to be thrown.

catch$[v^V]$**:** represents a catch block. It contains a node representing the first node of the catch block as a flow child. Parameter v^V is the node representing the value to be thrown.

3.2 Building IR

To determine control flow of the analyzed application, the information from value analysis is needed. Thus, the IR is built gradually during the analysis.

Initially, IR for the entry script of the application (typically `index.php`) is built. This IR contains caller nodes – the nodes corresponding to function, method, and constructor calls, script inclusions, and eval statements. Since at this point, the information needed to compute control flow from these nodes is not yet available, the control flow is initially directed to the nodes that follow the calls.

The control flow is then extended during static analysis. When processing a caller node, the analysis framework provides the first phase implementation with all information already computed by the analysis that is relevant to determine the control flow. Using this information, the first phase implementation finds appropriate function or method definitions or scripts to be included, and it computes IRs representing their control flow. The first phase implementation can build new IRs or use existing IRs, which are then shared among multiple caller nodes. This way, the first phase implementation can control context sensitivity. Finally, the control flow of the caller nodes is extended with computed IRs.

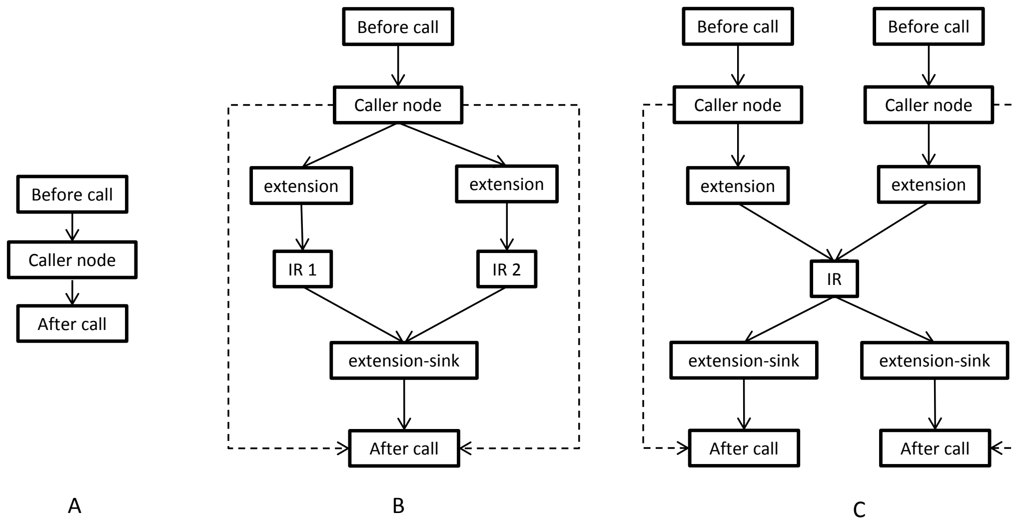


Figure 4 Building IR. Initial IR – the control flow of the caller node has not yet been extended (A). IR after processing the caller node during static analysis. The control flow of the caller node is extended with two IRs – IR 1 and IR 2 (B). Single IR shared between multiple caller nodes (C).

IRs are not connected to caller nodes directly – extension node is inserted between each caller node and the entry node of the connected IR and extension-sink node is inserted between each final node of the IR and the node following the call. While an extension node binds actual parameters to formal parameters for function, method, and constructor calls, an extension-sink joins states of final nodes of all the IRs that extend the corresponding caller node.

▶ **Example 2.** Fig. 4-A shows IR after it is initially built. Fig. 4-B shows IR after extending the caller node during the analysis. In this case, the caller is extended with more IRs – this can happen, e.g., if a method is called on an object that can be of more types. Fig. 4-C shows the case when a single IR is shared by multiple callers.

3.3 Analysis Domain

The states of our abstract domain have the form of $\overline{\text{State}} = \overline{H} \times \overline{V} \times \overline{F}$ where $\overline{H}$ is a state of the heap analysis, $\overline{V}$ is a state of the value analysis, and $\overline{F}$ is a state of the declaration analysis. The heap analysis tracks the shape of the heap and approximates concrete heap locations with heap identifiers (HId), while the value analysis tracks values on heap identifiers. While the heap analysis and value analysis need to interplay, the declaration analysis is independent from both.

Declaration Analysis

Declaration analysis is necessary, because in PHP and other dynamic languages, the names of functions, classes, and constants are bound to concrete definitions during runtime. The analysis thus needs to track these definitions. A state of a declaration analysis $\overline{F}$ is a set of class, function, and constant declarations and lattice operators of the analysis are $\langle \overline{F}, \subseteq, \cup, \cap \rangle$.

▶ **Example 3.** Consider the following PHP code:

```
1
2  if ($_GET[1]) {
3    class A {
4      public $a = 2;
5      function f($p) { return $p + 1;}
6    }
7  } else {
8    class A {
9      public $a = −1;
10     function f($p) { return $p − 1;}
11   }
12 }
13 $x = new A();
14 $y = $x−>f($x−>a); // $y can be −2, 0, 1, 3
```

Since the condition at line (1) is statically unknown, the declaration analysis computes that both declarations of class `A` can be used at line (12). Consequently, the call at line (13) can be done with two possible arguments and has two possible callees resulting in four possible results.

Heap Analysis

In PHP and other dynamic languages, variables as well as array indices and object properties need not be declared and can be accessed with arbitrary expressions, which can yield statically unknown values. If a specified variable, index, or object property exists, it is overwritten; if not, it is created.

To be able to capture this semantics, the heap analysis approximates arrays, objects, array indices, object fields, and even variables[2] with heap identifiers and the heap analysis can create new heap identifiers both during assignment and join operation. Whenever a new heap identifier is created, it is initialized with an existing heap identifier that stores values from statically unknown assignments to the new identifier that could happen before the creation.

Creation of new heap identifiers corresponds to *materialization* in shape analysis [20]. The *summary* heap identifiers summarize all the heap elements that could be updated by statically unknown assignments and have not been materialized yet[3]. When there is a need to distinguish a heap element from other heap elements summarized by the same summary heap identifier, a new heap identifier is materialized from the summary identifier. This happens, e.g., when an array index is assigned for the first time with a statically known target. In this case, the array index is approximated by the summary heap identifier representing all indices that could be updated only by statically unknown assignments in the pre-state and by the newly materialized heap identifier in the post-state. Materializations are defined as a set of pairs of heap identifiers $\mathrm{Mat} = \mathcal{P}(\mathrm{HId} \times \mathrm{HId})$. The meaning of a single materialization is that the first heap identifier from the pair is materialized from the second, summary heap identifier.

Note that materialization makes the naming scheme of the heap analysis flow-dependent – depending on the program location, a concrete heap element can be approximated by different heap identifiers. This makes an interplay of the heap analysis and the value analysis more

[2] Variables are treated as indices of a special associative array representing the symbol table.
[3] Heap elements that have not been assigned by any assignment are summarized by a special heap identifier `uninit`.

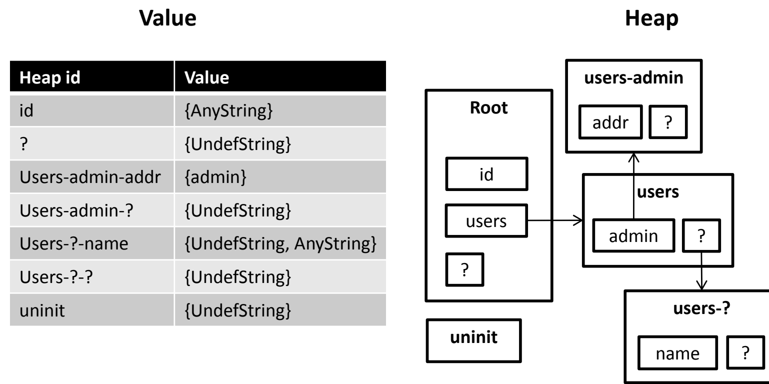


Figure 5 The heap and string part of the value component of the state after the update at line 23 in Fig. 1.

challenging. Since the value analysis tracks values on heap identifiers, materializations, which change the naming scheme, need to be applied also to value analysis. Later, we define this application using the standard abstract interpretation interface of the value analysis. This makes it possible to update the state of the value analysis automatically, without modifying the value analysis ad-hoc. That is, any value analysis that complies with the standard abstract interpretation interface can be used.

▶ **Example 4.** Fig. 5 shows the heap and value component of the analysis state after the update at line 23 in Fig. 1. In the following, we use the heap domain developed in [11] and the set domain as a value domain. Note that the value domain tracks values just over these heap identifiers that can contain values. Other heap identifiers are present only in the heap domain.

The heap component of the state contains heap identifier `Root` representing the array corresponding to the symbol table and heap identifier `uninit` representing the uninitialized heap elements. The symbol table array contains three heap identifiers (`id`, `users`, and `?`), which represent program variables (`$id` and `$users`) and statically unknown variables. For heap identifier `id`, value analysis tracks the value `AnyString`, while heap identifier `users` is present only in the heap domain and points to another array. Heap identifier `users-admin` represents index `$users[admin]`, while heap identifier `users-?` represents statically unknown indices of array `$users`. Both heap identifiers represent arrays corresponding to the next dimensions of array `$users`. Finally, heap identifiers `users-admin-addr`, `users-admin-name`, `users-admin-?`, `users-?-name`, and `users-?-?` represent indices of these arrays. Since these heap identifiers store values, they are tracked by the value analysis.

We assume that heap analysis is provided with lattice operators $\langle \overline{H}, \sqsubseteq_{\overline{H}}, \sqcup_{\overline{H}}, \sqcap_{\overline{H}} \rangle$. Operator $\sqsubseteq_{\overline{H}}$ specifies a partial order, $\sqcup_{\overline{H}}$ is the join operator, and $\sqcap_{\overline{H}}$ is the meet operator. The semantics of heap analysis is given by transfer function $[\![\cdot]\!]_{\overline{H}} : \overline{H} \mapsto \overline{H}$.

Moreover, we assume that the heap analysis provides function read : $AE \mapsto \mathcal{P}(\text{HId})$ for reading data from the heap. The function returns a set of heap identifiers identified by given *access expression*. Access expression is obtained from IR nodes of type variable^V, property-use^V, and index-use^V. In the case of variable^V, access expression is just the set of values, in the case of property-use^V, and index-use^V, it is a sequence of sets of values. Each set from the sequence contains values that can be used to access the corresponding dimension of an array or the corresponding object in the object reference chain. That is,

access expressions can represent multi-dimensional updates. This is necessary in order to model semantics of non-decomposable multi-dimensional updates [11].

▶ **Example 5.** Consider reading the values stored at index `$users[10]['name']` from the state depicted in Fig. 5. The access expression for the index is $\{users\}$ $\{10\}$ $\{'name'\}$. Calling the read function provided by the heap component with this access expression as argument returns the heap identifier `users-?-name`. This heap identifier is then used to get the resulting values from the value domain. The value domain returns values `UndefString` and `AnyString` meaning that the index `$users[10]['name']` can be uninitialized and can contain statically unknown string value.

Similarly, when reading the index `$users[$_GET[1]]['name']`, the access expression is $\{users\}\{*\}\{'name'\}$, the read function returns heap identifiers `users-?-name` and `users-admin-name`, and the subsequent call to the value domain returns values `UndefString`, `AnyString`, and `'addr'`.

We additionally assume that the heap analysis provides function joinToValue : $\overline{H} \times \overline{H} \mapsto \mathrm{Mat} \times \mathrm{Mat}$. This function takes the heap parts of analysis states to be joined as arguments and for each joined analysis state, it returns materializations of the heap identifiers created when performing the join operation.

Finally, we assume that heap analysis provides function assignToValue : $\overline{H} \times \mathrm{AE} \mapsto \mathrm{Mat} \times \mathcal{P}(\mathrm{HId}) \times \mathcal{P}(\mathrm{HId})$. This function takes an analysis state before the assignment and the access expression identifying the target of the expression as arguments and returns a triple: (i) materializations of heap identifiers created when performing the assignment, (ii) the heap identifiers representing heap elements that certainly must be updated, and (iii) the heap identifiers representing heap elements that only may be updated.

Value Analysis

The states of the value analysis have a form of $\overline{V} = \overline{V_1} \times \overline{V_2}$ where $\overline{V_1}$ is a state of the value analysis in the first phase and $\overline{V_2}$ is a state of the value analysis in the second phase (end-user analysis). The first phase of the analysis modifies the first component of the state V_1, the second phase of the analysis modifies the second component of the state V_2.

The user of the framework can define both the value analysis in the first phase and the value analysis in the second phase. However, since values that are used to compute control-flow and targets of data accesses are computed in the first phase, the user is more constrained in the first phase.

Second phase

The domain for the second phase tracks information over heap identifiers and it is provided with lattice operators $\langle \overline{V_2}, \sqsubseteq_{\overline{V_2}}, \sqcup_{\overline{V_2}}, \sqcap_{\overline{V_2}} \rangle$, transfer function $[\![\cdot]\!]_{\overline{V_2}} : \overline{V_2} \mapsto \overline{V_2}$, and widening operator $\nabla_{\overline{V_2}}$.

First phase

In the first phase, the value analysis tracks values of PHP primitive types over heap identifiers:

$$\overline{V_1} = \mathrm{HId} \mapsto \overline{\mathrm{Value}_1}$$
$$\overline{\mathrm{Value}_1} = \overline{\mathrm{Undef}} \times \overline{\mathrm{Null}} \times \overline{\mathrm{Bool}} \times \overline{\mathrm{Num}} \times \overline{\mathrm{String}}$$

Since PHP has dynamic type system – variables, array indices, and object properties do not have declared types, and they can store values of different types depending on context –, $\overline{\text{Value}_1}$ can store values of all primitive types.

To define the value analysis of the first phase, the user of the framework must for each component $\overline{C}$ of $\overline{\text{Value}_1}$ provide the framework with lattice operators $\langle \overline{C}, \sqsubseteq_{\overline{C}}, \sqcup_{\overline{C}}, \sqcap_{\overline{C}} \rangle$, transfer function $[\![\cdot]\!]_{\overline{C}} : \overline{C} \mapsto \overline{C}$, and widening operator $\nabla_{\overline{C}}$. The lattice operators for $\overline{\text{Value}_1}$ are defined component-wise. Moreover, for each pair $(\overline{C_1}, \overline{C_2})$ of components of $\overline{\text{Value}_1}$, functions $\text{Conv}_{\overline{C_1 C_2}} : \overline{C_1} \mapsto \overline{C_2}$ and $\text{Conv}_{\overline{C_2 C_1}} : \overline{C_2} \mapsto \overline{C_1}$ must be provided. These functions are used to model type conversions, which are ubiquitous in dynamic languages.

It should be noted that even though value analysis in the first phase is defined independently of heap analysis, which simplifies its definition, intricate value semantics of PHP makes the definition inherently complex. The framework thus provides default implementations of all components of $\overline{\text{Value}_1}$ including transition functions for PHP native operators and library functions. For the default implementation of the numeric component, we used the interval domain. For the default implementation of the string component, we used the domain based on sets of strings. Its lattice structure is $\langle \mathcal{P}(\text{String}), \subseteq \rangle$ where String are concrete strings. To make the height of the lattice finite and thus guarantee termination, the size of sets is limited by a constant; value AnyString represents the sets of larger sizes.

▶ **Example 6.** This example illustrates the states of $\overline{\text{Value}_1}$ with the default implementation of its components.

The abstract value $(\bot, \bot, \text{AnyBool}, \bot, \bot)$ represents concrete values `true` and `false` of type Boolean, the abstract value $(\text{undef}, \bot, \bot, \text{true}, \{\text{"foo"}, \text{"bar"}\})$ represents the following concrete values: uninitialized value, the Boolean `true`, the string `"foo"`, and the string `"bar"`.

3.4 Lattice Order and Meet

The lattice order $\sqsubseteq_{\overline{\text{State}}}$ and meet operator $\sqcap_{\overline{\text{State}}}$ for analysis states are defined component-wise:

$$(\overline{h_1}, \overline{v_1}, \overline{f_1}) \sqsubseteq_{\overline{\text{state}}} (\overline{h_2}, \overline{v_2}, \overline{f_2}) \iff h_1 \sqsubseteq_{\overline{H}} \overline{h_2} \land \overline{v_1} \sqsubseteq_{\overline{V}} \overline{v_2} \land \overline{f_1} \subseteq \overline{f_2}$$
$$(\overline{h_1}, \overline{v_1}, \overline{f_1}) \sqcap_{\overline{\text{state}}} (\overline{h_2}, \overline{v_2}, \overline{f_2}) = (\overline{h_1} \sqcap_{\overline{H}} \overline{h_2}, \overline{v_1} \sqcap_{\overline{V}} \overline{v_2}, \overline{f_1} \cap \overline{f_2})$$

3.5 Applying Materializations to Value Analysis

Materializations allow the heap analysis to create new heap identifiers during the assignment and join operations. As discussed in Sect. 3.3, materializations change the naming scheme of the heap analysis; since value analysis tracks values of heap identifiers, these changes must be applied also to the value domain. This is carried out by function $\text{applyMaterializations} : (\overline{V} \times \text{Mat}) \mapsto \overline{V}$ that applies materializations to a state of the value analysis:

$$\text{applyMaterializations}(\overline{v}, M) = \overline{v_n} \textit{ where } M = \{(t_1, s_1), (t_2, s_2), ..., (t_n, s_n)\},$$
$$\overline{v_0} = \overline{v}, \forall j \in [1..n] : \overline{v_j} = [\![t_j = s_j]\!]_{\overline{V}}(\overline{v_{j-1}})$$

3.6 Join and Widening

The join of two facts is defined as the set of all facts that are implied independently by any. The join and widening of two states (h_1, v_1, f_1) and (h_2, v_2, f_2) are defined as follows:

$$(\overline{h_1}, \overline{v_1}, \overline{f_1}) \sqcup_{\overline{\text{state}}} (\overline{h_2}, \overline{v_2}, \overline{f_2}) = (\overline{h_1} \sqcup_{\overline{H}} \overline{h_2}, \overline{v_1'} \sqcup_{\overline{V}} \overline{v_2'}, \overline{f_1} \cup \overline{f_2})$$
$$(\overline{h_1}, \overline{v_1}) \nabla_{\overline{\text{state}}} (\overline{h_2}, \overline{v_2}) = (\overline{h_1} \sqcup_{\overline{H}} \overline{h_2}, \overline{v_1'} \nabla_{\overline{V}} \overline{v_2'}, \overline{f_1} \cup \overline{f_2})$$
$$(\overline{m_1}, \overline{m_2}) = \text{joinToValue}(\overline{h_1}, \overline{h_2})$$
$$\overline{v_1'} = \text{applyMaterializations}(\overline{v_1}, \overline{m_1})$$
$$\overline{v_2'} = \text{applyMaterializations}(\overline{v_2}, \overline{m_2})$$

The declaration and heap parts of input states are joined independently on other parts. To perform the join of value parts, heap analysis provides value analysis with materializations of heap identifiers in each joined state. Then, the materializations are applied to the value components of joined states and finally, the updated value parts are joined. Note that the latter two operations are done just by means of standard abstract interpretation interface provided by value analysis.

▶ **Example 7.** Fig. 6 shows joining value and heap components of two states $(\overline{v_1}, \overline{h_1})$ and $(\overline{v_2}, \overline{h_2})$. For brevity we omit declaration components.

First, the heap components of analysis states are joined. For the first state to be joined, heap analysis materializes heap identifiers `arr-1-3`, `arr-1-?`, and `arr-?-?` from the heap identifier `uninit` representing undefined heap identifier. That is, there were no statically-unknown assignments that could update the materialized identifiers. Application of materializations to the value component of the first analysis state to be joined thus just adds the materialized identifiers and initializes them with value `UndefString`.

For the second state, heap analysis materializes heap identifiers `arr-1-?`, `arr-1-2`, and `arr-1-3`. Since there was statically-unknown assignment that could update the latter identifier, this identifier is materialized from the identifier `arr-?-3` representing the target of this statically-unknown assignment. Application of materializations to the value component of the second analysis state to be joined thus initializes the identifier `arr-1-3` with values `UndefString` and `'second'`.

Finally, the value components of analysis states after applying materializations $\overline{v_1'}$ and $\overline{v_2'}$ have the same set of heap identifiers and can be joined independently on the heap components.

3.7 Transfer Functions

For each kind of node in the intermediate representation, a transfer function maps an abstract state before the node to an abstract state after the node.

We describe the transfer function for the node $\text{assign}^V[l^V, r^V]$, where both parameters l^V and r^V are nodes of type variable^V, property-use^V, or index-use^V. Each of these nodes allows getting an access expression, which provides heap analysis information necessary for accessing heap identifiers representing heap locations stored in the node. The access expression for the left-hand side of the assignment l^V is $l^V.\text{AE}$, the access expression for the right-hand side of the assignment r^V is $r^V.\text{AE}$.

To define the transfer function, we first define function strongUpdate : $\overline{V} \times \mathcal{P}(\text{HId}) \times \mathcal{P}(\text{HId}) \mapsto \overline{V}$. The first parameter is a state of value analysis that is being updated, the second parameter is the set of heap identifiers that are updated, and the last parameter is

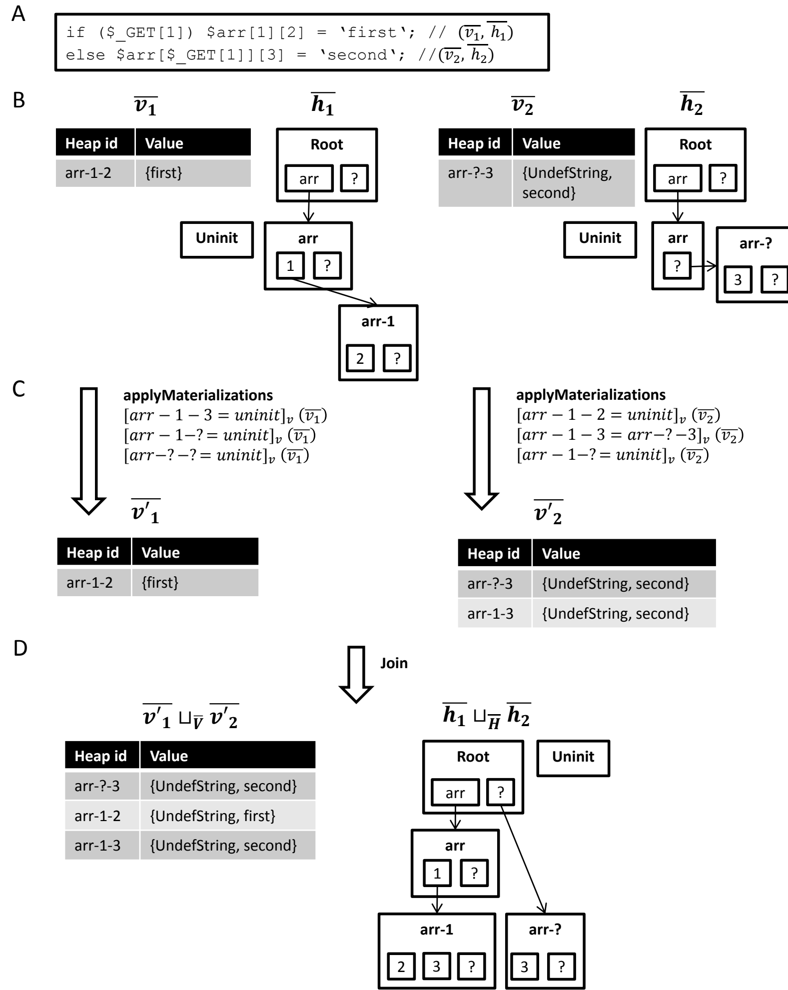


Figure 6 Joining value and heap components of two states $(\overline{v_1}, \overline{h_1})$ and $(\overline{v_2}, \overline{h_2})$. The corresponding code (A), value and heap components of joined states (B), applying materializations to value components of states to be joined (C), result of the join (D). For the sake of space, the heap identifiers that have just value `UndefString` are not depicted in value components.

the set of heap identifiers representing new values:

$$\text{strongUpdate}(\overline{v}, T, S) = \overline{v_n} \ \textit{where}\ T = \{t_1, t_2, ..., t_n\}, \overline{v_0} = \overline{v}$$

$$\forall j \in [1..n] : \overline{v_j} = \bigsqcup_{s \in S} [\![t_j = s]\!]_{\overline{V}}(\overline{v_{j-1}})$$

Next, we define function weakUpdate : $\overline{V} \times \mathcal{P}(\text{HId}) \times \mathcal{P}(\text{HId}) \mapsto \overline{V}$:

$$\text{weakUpdate}(\overline{v}, T, S) = \bigsqcup_{t \in T, s \in S} \overline{v} \sqcup_{\overline{V}} [\![t = s]\!]_{\overline{V}}(\overline{v})$$

While after strong update, heap identifiers can have just new values, after weak update, they either can have the original values or the new ones [18]. This effect is approximated by joining the analysis state before the update with the analysis state after the update.

The transfer function for updating the state $(\overline{h}, \overline{v})$ with $\text{assign}^V[l^V, r^V]$ is defined as:

$$[\![\text{assign}^V[l^V, r^V]]\!]_{\overline{\text{state}}}(\overline{h}, \overline{v}, \overline{f}) = ([\![l^V.\text{AE} = r^V.\text{AE}]\!]_{\overline{H}}(\overline{h}), \overline{v'''}, \overline{f})$$
$$(m, u_{must}, u_{may}) = \text{assignToValue}(\overline{h}, l^V.\text{AE})$$
$$\overline{v'} = \text{applyMaterializations}(\overline{v}, m)$$
$$\overline{v''} = \text{strongUpdate}(\overline{v'}, u_{must}, \text{read}(\overline{h}, r^V.\text{AE}))$$
$$\overline{v'''} = \text{weakUpdate}(\overline{v''}, u_{may}, \text{read}(\overline{h}, r^V.AE))$$

The transfer function for the heap part of the state is defined by the heap domain itself, and it is not influenced by the value domain. To define the transfer function for the value part of the state, the heap domain provides the value domain with necessary information via function assignToValue. This information consists of: (1) m – information necessary to materialize the heap identifiers that were defined by the assignment (2) heap identifiers representing the heap elements that are certainly targets of the assignment, and (3) heap identifiers representing the heap elements that may be targets of the assignment.

Then, the materializations are applied to the value domain. Finally, the heap identifiers that are certainly targets of the assignment are strongly updated with values of the heap identifiers read from the right-hand side of the assignment, and the heap identifiers that only may be targets of the assignment are weakly updated. The same way as in the case of the join operation, all these updates are performed just by means of the transfer function for the assignment provided by value analysis.

▶ **Example 8.** Fig. 7 illustrates the transition function for the assignment at line 24 in Fig. 1 (`$users[$id]['addr'] = $_GET['addr']`). First, the access expressions for the source and the target of the assignment are obtained from the corresponding IR nodes. For the source of the assignment, the access expression is $\{_GET\}\{addr\}$, for the target of the assignment, the access expression is $\{users\}\{AnyString\}\{addr\}$. Note that in the latter case, the value for the second dimension of the access is specified by the variable `$id`.

Second, the access expressions are used to specify the update. During the update, the heap component materializes the heap identifier `users-?-addr` and this change is propagated to the value component via the function applyMaterializations. Note that since there have not been any statically unknown assignments that could update this heap identifier, it is materialized from the identifier `uninit` representing undefined values. That is, the identifier `users-?-addr` is added to the value component and initialized with `UndefString`.

Finally, the heap component specifies that identifiers `users-?-addr` and `users-admin-addr` are weakly updated and the update is propagated to the value component. Since the target of the assignment is not statically known, no heap identifiers are strongly updated.

3.8 Summary Heap Identifiers

Value analyses are designed to track information on local variables, while we use them to track information on heap identifiers, which can represent many concrete heap locations – summary identifiers [9]. For an example of summary identifiers, consider heap identifiers representing targets of statically unknown assignments and heap identifiers representing a single allocation-site in that many concrete heap locations can be allocated. While value analysis can treat heap identifiers that represent a single heap location exactly the same way as local variables, for summary identifiers, it must take into account that they represent more heap locations.

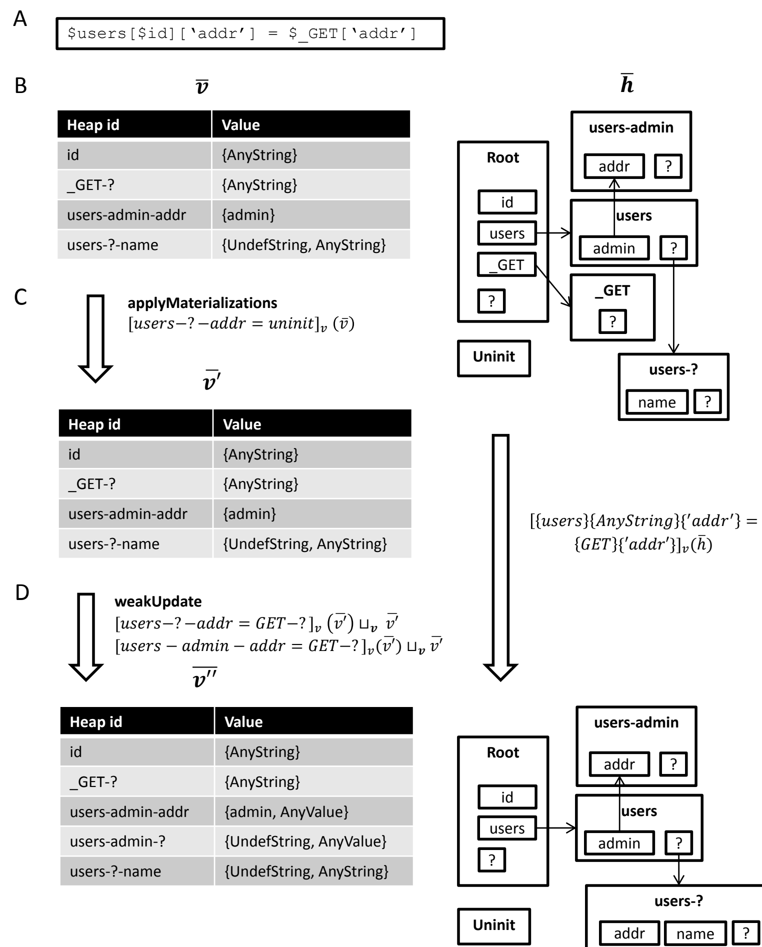


Figure 7 Transfer function for the assignment. The code of the assignment (A). The value and the heap component $(\overline{v}, \overline{h})$ of the state before the assignment (B). Applying materialization of the identifier `users-?-addr` to the value component of the state (C). The value component $\overline{v''}$ and the heap component of the state after the assignment (D). For the sake of space, the heap identifiers that have just value `UndefString` are not depicted in value components.

First, summary heap identifiers must be always weakly updated. In our framework, heap analysis has to take this into account in function assignToValue, which defines identifiers that are weakly and strongly updated by the assignment. This is enough for non-relational value domains – these value domains can otherwise treat summary heap identifiers the same way as local variables.

However, in the case of relational value domains, it is additionally necessary to treat differently assignments from summary heap identifiers. Consider the code:

```
1  1
2  $a = $users[$_GET[1]];
3  $b = $users[$_GET[2]];
4  if ($a != $b) {...}
```

Our heap analysis represents both `$users[$_GET[1]]` and `$users[$_GET [2]]` by the same summary heap identifier `users-?`. Our technique would thus abstract the semantics of assignment at line (1) as $[\![a = users{-}?]\!]_{\overline{V}}$ and the semantics of assignment at line (2) as

$[\![b = users{-}?]\!]_{\overline{V}}$. If v werte a relational domain, the analysis would relate both identifiers `a` and `b` with the summary identifier `users-?` and incorrectly infer that the `if-then` branch can never be reached. This problem was studied by Gopan [9] et. al., who showed that it is wrong to correlate summarized identifiers with non-summarized ones and they proposed a way to extend existing relational domains to deal with this problem.

In our framework, the value domain in the first phase is non-relational and all value domains for end-user analyses that we have implemented so far are also non-relational. To use relational value analyses, these analyses need to be extended to summary dimensions as described by Gopan [9] and the heap analysis has to additionally provide the framework with the information which heap identifiers are summaries.

3.9 Soundness

Our analysis framework allows for defining sound analyses. If the semantics of heap analysis, the semantics of value analysis, and the semantics of declaration analysis plugged into our framework are sound, the semantics of the resulting composed analysis is sound as well. In the following, we will state the fundamental assumptions on value and heap analyses[4]. The traditional soundness argument in abstract interpretation is:

▶ **Definition 9** (Soundness of analysis semantics)**.** Given a set of abstract states $\overline{S}$, abstract semantics $[\![\cdot]\!]_{\overline{S}}$, a set of concrete states S, concrete semantics $[\![\cdot]\!]_S$, and concretization function $\gamma : \overline{S} \mapsto \mathcal{P}(S)$, the abstract semantics $[\![\cdot]\!]_{\overline{S}}$ is sound with respect to the concrete semantics $[\![\cdot]\!]_S$ iff for each statement st and analysis state $\overline{s} \in \overline{S}$ it holds:

$$([\![st]\!]_{\overline{S}}(\overline{s}) = \overline{s}' \wedge [\![st]\!]_S(\gamma(\overline{s})) = s') \implies s' \subseteq \gamma(\overline{s}')$$

Hence, to prove the soundness of the analysis semantics, it is necessary to define the structure of concrete states, their semantics, concretization function, and then prove that it satisfies proposition of Def. 9.

The soundness argument is based on the assumption that the heap semantics and the value semantics are sound. While for the value semantics, the soundness can be specified just using Def. 9 and we can thus use any sound value analysis in our framework, for the heap semantics, we must pose further assumptions.

First, we assume that function read provided by heap analysis complies with the semantics of concrete dereferencing. That is, for each abstract state and each access expression, the heap identifiers returned by function read must represent all concrete locations given by dereferencing using the access expression in all the concretizations of the abstract state.

Next, we assume that the updates given by semantics function assignToValue are sound with respect to the semantics of concrete dereferencing. That is, for the left hand site of assignment, the heap identifiers representing updates given by function assignToValue and an access expression in an abstract state must represent all concrete heap locations R given by dereferencing using the access expression in all the concretizations of the abstract state. Moreover, all the heap identifiers that are in the strong-update set (u_{must}) must exactly represent all the heap locations in set R and all the other heap identifiers that represent the sets of heap locations with non-empty intersection with R must be in the may-update set (u_{may}).

[4] We will omit the declaration analysis. It needs not interplay with the other analyses and can be treated completely separately.

Finally, we assume that the materializations produced by heap analysis are coherent with respect to the modifications of heap analysis. That is, (1) in the post-state, the heap identifiers that are not sources of materializations must represent the same concrete heap locations as in the pre-state, (2) for each heap identifier that is materialized and its source it must hold that in the post-state each of them represents the subset of the heap locations represented by the source of the materialization in the pre-state, and (3) in the post-state both heap identifiers together must represent all the heap locations represented by the source of the materialization in the pre-state.

Note that we do not require different heap identifiers to represent non-overlapping portions of concrete heap. That is, there can exist two different heap identifiers with *overlapping concretizations*, i.e., there exists a concrete heap location approximated by both heap identifiers. This allows using heap analyses modeling the semantics of assignment by reference more precisely [11]. Consider, e.g., the statement `$a = &$b`. In the concrete semantics, the statement makes `$a` and `$b` pointing to the same heap location. We allow heap analysis to model this concrete location by more heap identifiers with overlapping concretizations – e.g., heap identifier i_1 for variable `$a` and heap identifier i_2 for variable `$b`. To be sound, heap analysis must update these heap identifiers coherently – e.g., if heap identifier i_1 is updated, heap identifier i_2 is updated as well. This is guaranteed by the soundness of updates stated above. Heap identifiers with overlapping concretizations can enable more strong updates. Consider the following example:

```
1 if ($_GET['INPUT']) $a = &$b;
2 else $a = &$c;
3 $a = 1;
```

There are two concrete heap locations in this example. If heap analysis uses less than three heap identifiers to represent these concrete locations, it must perform weak update at line 3. In case of three heap identifiers, their concretizations must overlap; it allows heap analysis to perform strong update on the heap identifier for `$a` and weak updates of those for `$b` and `$c`.

4 Evaluation

To evaluate the precision and scalability of our framework, we used it to implement static taint analysis and we applied it to the NOCC webmail client[5] and a benchmark application comprising of a fragment of the myBloggie weblog system[6], with a total of over 16 kLoC. The benchmark application contains 13 security problems; the number of problems contained in the webmail client is not known.

We compared the results of our analysis with Pixy [15] and Phantm [17], the state-of-the-art tools for security analysis and error discovery in PHP applications. Both these tools compute control-flow of analyzed applications, model PHP data structures, and perform value analysis. Both these tools detect accesses to uninitialized elements. In addition, Phantm detects type mismatch errors and Pixy detects taint errors, i.e., flows of sensitive data to critical commands. Our analysis detects both type of errors.

Tab. 2 shows the summary of results. Out of 13 errors in the benchmark application, 8 errors were accesses to uninitialized elements and 5 errors were taint errors. Since Pixy is not designed to detect taint errors we did not use taint errors to assess the error coverage

[5] `http://nocc.sourceforge.net/`
[6] `http://mybloggie.mywebland.com/`

Table 2 Comparison of tools for static analysis of PHP. W/C/F/T: **W**arnings / error **C**overage (in %) / **F**alse-positives rate (in %) / analysis **T**ime (in s).

	myBloggie				**NOCC 1.9.4**			
Lines	648				15,605			
	W	**C**	**F**	**T**	**W**	**C**	**F**	**T**
Our framework	16	100	19	0.9	34	NA	62	84
Pixy	16	69	44	0.6	NA			
Phantm	43	38	93	2.5	426	NA	NA	90

for Pixy. The table shows that the analysis defined using our framework outperforms the other tools both in error coverage and number of false positives when analyzing the benchmark application. As to the analysis of NOCC, while Pixy was even not able to analyze the application, Phantm reported a huge number of alarms, which together with a high false-positive rate made its output almost useless[7].

Our analysis discovered all 13 problems in myBloggie. One of the false alarms reported by our analysis is caused by imprecise modeling of the built-in function `date`. Our analysis only models this function by types and deduced that any string value can be returned by this function. However, while the first argument of the function is `"F"`, the function returns only strings corresponding to English names of months. When the value returned by this function is used to access the index of an array, our analysis incorrectly reports that an undefined index of the array can be accessed. Two remaining false alarms are caused by path-insensitivity of the analysis. The sanitization and sink commands are guarded by the same condition, however, there is a joint point between these conditions, which discards the effect of sanitization from the perspective of path-insensitive analysis.

Our analysis found three previously unknown vulnerabilities in the NOCC email client and ten other problems (e.g., calling a function with an argument that is not declared and superfluous implicit conversions). False-positive alarms were caused by imprecise modeling of PHP built-in functions, path-insensitivity of the analysis, and using non-relational value domains.

5 Related Work

The present work builds on a large body of work on static program analysis of dynamic languages. The pioneering works [12, 30, 26, 27] omit modeling some of the important parts of the analyzed languages. The unmodeled parts include references, dynamic accesses to associative arrays, and object-oriented features.

Pixy [16] performs security taint analysis of PHP programs and provides information about the flow of tainted data. Pixy performs a flow-sensitive, interprocedural, and context-sensitive data flow analysis along with literal and alias analysis to achieve precise results. Its main limitations include an incomplete support for statically-unknown updates to associative arrays, ignoring classes and the `eval` command, omitting type inference, and a limited support for handling file inclusion and aliasing. Alias analysis introduced in Pixy incorrectly models aliasing when associative arrays and objects are involved.

[7] Because of a huge number of alarms reported by Phantm, we assessed its false-positives rate just for myBloggie, not for NOCC.

Andromeda static taint analyzer [24] fights the problem of scalability of taint analysis by computing data-flow propagations on demand. It uses forward data-analysis to propagate tainted data and ignores propagation of other data. If tainted data are propagated to the heap, it uses backward analysis to compute all targets to which the data should be propagated. Andromeda analyzes Java, .NET, and JavaScript applications. The drawback of the approach is that it propagates only taint information. Especially for dynamic languages, the control-flow of the application can depend on other kind of information which is then not available. To reduce this problem, Andromeda uses F4F [21], which reduces the amount of information that is not known statically.

Phantm [17] is a PHP 5 static analyzer for type mismatch based on data-flow analysis; it aims at detection of type errors. To obtain precise results, Phantm is flow-sensitive, i.e., it is able to handle situations when a single variable can be of different types depending on program location. However, it omits updates of associative arrays and objects with statically-unknown values and aliasing, which can lead to both missing errors and reporting false positives.

TAJS [14] is a JavaScript static program analysis infrastructure that infers type information. To gain precise results, the analysis is context-sensitive and precisely models intricate semantics of JavaScript, including prototype objects and associative arrays, dynamic accesses to these data structures, and implicit conversions. It tackles the problem that dynamic features of JavaScript make it impossible to construct control-flow before static analysis by constructing control-flow on-the-fly during the analysis. Since TAJS models JavaScript semantics precisely, it has been successfully used to enable additional analyses. In [4, 5], the TAJS program analysis infrastructure is used to build a tool for refactoring JavaScript programs and in [13] TAJS is used to enable technique of statically resolving `eval` constructs. However, TAJS combines heap and value (type) analysis ad-hoc, which results in intricate lattice structure and transfer functions. Next, TAJS assumes that updates to multi-dimensional arrays and objects can be decomposed to updates of length one. While this is true for JavaScript, this assumption leads to loss of precision in the case of some other dynamic languages such as PHP and Perl.

Since the excess of information that are only available at runtime pose a major problem to static analysis, several techniques have been developed that try to enable static analysis of dynamic languages by making this information statically available prior to static analysis. F4F [21] focuses on static taint analysis of web applications that use frameworks. They use a semi-automatically generated specification of framework-related behaviors to reduce the amount of statically-unknown information, which arises, e.g., from reflective calls. Phantm [17] reduces the number of information that static analysis must compute and possibly overapproximate by first executing the application, collecting this information and then invoking static analysis from a particular runtime state. Wei et. al. [28] reduce the number of statically-unknown information in JavaScript by using a technique of blended static analysis [2]. They first execute a test suite and for each test they record its execution trace. Then, for each execution trace, they extract its call graph, types of created objects, and dynamically generated code and perform static analysis of the application with using this information. Finally, they combine solutions from different execution traces into a single solution for the application.

Recently, there has flourished a rich body of work on precise and sound points-to analysis for dynamic languages. Sridharan et. al. [22] present static flow-insensitive points-to analysis for JavaScript modeling objects in JavaScript using associative arrays that can be accessed by arbitrary expressions. To enhance the precision and scalability of the analysis, they identify

correlations between dynamic property read and write accesses. If the updated location and stored value can be accessed by the same first class entity (variable), it is extracted to a function parametrized by this entity; this function is then analyzed context-sensitively with the context being the variable. Thus, the correlation between the update and store is preserved. Wei et al. [29] present points-to analysis for JavaScript. Their analysis is partially flow-sensitive – it stores points-to information for every CFG segment with a single state-update statement. Next, their analysis is context-sensitive – to reflect the fact that properties can be added to objects at runtime, it uses a receiver object, its chain of prototype objects, its local properties and their object values as a context. This makes it possible to differentiate between two calls received by the object with the same creation site but different properties. Finally, to perform more strong updates in case of property-writes they use access path edges in their points-to representation. For a property-write (e.g., `$x->p = $y`) where the dereferenced variable (`$x`) points to more objects, weak-updates of properties in these objects (`p`) are performed. However, the access path edge (`<x, p>`) is strongly updated. If the property is read and there exists a corresponding access path edge, it is used instead of points-to edges (for `$z = $x->p` the access path edge `<x, p>` is used and just objects pointed-to by variable `$x` are read). In our previous work [11], we presented points-to analysis for PHP modeling associative arrays that could be accessed using arbitrary expressions. Additionally, our analysis precisely models the semantics of PHP explicit aliases and the semantics of multi-dimensional updates – in PHP or Perl, updates create indices if they do not exist and initialize them with empty arrays if needed; on contrary, read accesses do not.

While heap and value static analyses have been studied mainly as orthogonal problems, to support verification of real programs, they usually need to be combined together [6, 25]. Since in dynamic languages, data structures can be dynamically accessed with arbitrary expressions, this problem of combining heap and static value analysis is particularly relevant in this domain.

Clousot [3] preprocesses the program applying heap analysis, and uses a value numbering algorithm to compute under-approximation of must-alias to replace heap accesses with heap identifiers. Value analysis then tracks values of variables and also of the heap identifiers. While the approach allows for using arbitrary value analysis, it only allows for using specific heap analysis, which cannot use the information provided by value analysis, and the technique is not sound.

Miné et. al. [19] combine type based pointer analysis and numeric value analyses in a generic way. The pointer analysis models pointer arithmetic, union types and records of stack variables in C programs. The general limitation of this technique is that it relies on type based heap analysis, which is too coarse for many applications. In particular, their technique does not support summary nodes and dynamic allocation.

Fu [8] combines numeric value analysis and points-to analysis. His method uses points-to analysis to partition possibly infinite set of heap references into a finite set of abstract locations (heap identifiers) and use value analysis to track values of variables and also of heap identifiers. The method is both generic – it allows for reusing existing analyses as black-boxes – and automatic – it does not require any annotations specific to a particular heap and value analysis to be provided. The fundamental limitation of the technique is that it relies on flow-independent naming scheme for points-to analysis. That is, a concrete reference is always mapped to the same abstract location independently of program location. On one hand, this assumption allows the technique to assume that change of the heap component of the analysis state has no effect on the value component of the state and that two states can

be joined component-wise. On the other hand, this assumption poses a substantial limitation to modeling of adding new object fields and array indices using statically-unknown updates. To illustrate the limitation, consider that a statically-unknown index of an empty array `$a` is updated (`$a[rand()]=..`). At this point, points-to analysis must represent all concrete indices of the array with a single abstract location h. Next, if a concrete index of the array, e.g., `$a[1]`, is updated (`$a[1]=..`), the analysis must still represent the index `$a[1]` with h and thus cannot distinguish this index from other indices in `$a`. That is, a statically unknown update makes the updated array (object) index-insensitive (field-insensitive) for all indices (fields) added after the update. As both modeling statically-unknown updates and field-sensitivity of heap analysis is crucial for static analysis of dynamic languages [23, 29], the assumption of flow-independent naming scheme is too limiting in this context.

Ferrara [6] introduced the concept of substitutions overcoming the limitation of flow-indepenent naming scheme when combining value and heap analysis. Substitutions allow heap analysis to materialize and summarize abstract locations, i.e., to replace a single abstract location in the pre-state with more abstract locations in the post-state and to replace more abstract locations in the pre-state with a single abstract location in the post-state. Ferrara defined how the analyses are composed when the substitutions are given and showed the assumptions on the heap and the value analyses in order to make their composition sound. However, his work cannot be directly applied in the context of dynamic languages. First, it does not model dynamically added fields and indices to objects and arrays, which is essential for dynamic languages. Then, Ferrara allows only heap analyses with non-overlapping heap identifiers. As we explain in Section 3.9, some heap analyses developed for dynamic languages use overlapping heap identifiers to perform more strong updates. Moreover, his work does not allow heap analyses to explicitly specify which updates are strong and which updates are weak thus reducing the precision of the composed analysis. In our work we focused specifically on heap analyses for dynamic languages overcoming these limitations.

6 Conclusion

In this paper, we presented a framework for static analysis of dynamic languages, in particular PHP applications.

The framework employs a two-phase analysis architecture – in the first phase, the dynamic constructs present in the analyzed code are resolved, while the analysis in the second phase can proceed in a way similar to a one for a language without dynamic features. This way, the framework provides a developer with high-level API for implementing various kind of analyses upon the code without the need to cope with dynamic features of the language. To allow resolving dynamic features, the framework combines heap, value, and declaration analyses. We described the necessary requirements on these analyses and the way these analyses are composed together generically and soundly. That is, our framework allows for combining various heap and value analyses while guaranteeing that if the analyses being composed are sound, the composed analysis is sound as well.

The framework is provided with default implementations of heap analyses and first phase analyses. To demonstrate usefulness of our framework, we implemented taint analysis of PHP application and applied it on real PHP application. We have shown that the tool is able to reveal real (previously unknown) security holes, while producing less false-positive alarms comparing to other state-of-the-art tools.

As for future work, we aim at improving the performance and precision of the analyzes provided by the framework especially in terms of scaling to large applications. In particular,

this includes the scalability improvements of the heap analysis, implementation of more choices of context-sensitivity, and devising precise modeling of more library functions. Next, we plan to enhance our implementation of security analysis and use the framework for implementing additional end-user analyses.

References

1 P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.

2 Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA'07*, pages 118–128. ACM, 2007.

3 Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS'10*, LNCS, pages 10–30. Springer-Verlag, 2011.

4 Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for javascript. In *OOPSLA'11*, pages 119–138. ACM, 2011.

5 Asger Feldthaus and Anders Møller. Semi-automatic rename refactoring for javascript. In *OOPSLA'13*, pages 323–338. ACM, 2013.

6 Pietro Ferrara. Generic combination of heap and value analyses in abstract interpretation. In *VMCAI'05*, LNCS, pages 302–321. Springer-Verlag, 2014.

7 Seth Fogie, Jeremiah Grossman, Robert Hansen, Anton Rager, and Petko D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense.* Syngress, May 2007.

8 Zhoulai Fu. Modularly combining numeric abstract domains with points-to analysis, and a scalable static numeric analyzer for java. In *VMCAI'05*, LNCS, pages 282–301. Springer-Verlag, 2014.

9 Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In *TACAS'04*, LNCS, pages 512–529. Springer-Verlag, 2004.

10 David Hauzar and Jan Kofroň. WEVERCA. `http://d3s.mff.cuni.cz/projects/formal_methods/weverca/`, 2014.

11 David Hauzar, Jan Kofroň, and Pavel Baštecký. Data-flow analysis of programs with associative arrays. In *ESSS'14*, EPTCS, pages 56–70. Open Publishing Association, 2014.

12 Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW'04*, pages 40–52. ACM, 2004.

13 Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *ISSTA 2012*, pages 34–44. ACM, 2012.

14 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *SAS'09*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

15 Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP'06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

16 Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP'06*, pages 258–263. IEEE Computer Society, 2006.

17 Etienne Kneuss, Philippe Suter, and Viktor Kuncak. Runtime instrumentation for precise flow-sensitive type analysis. In *RV'10*, LNCS, pages 300–314. Springer-Verlag, 2010.

18 Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *POPL'11*, pages 3–16, New York, NY, USA, 2011. ACM.

19 Antoine Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *LCTES'06*, pages 54–63. ACM, 2006.
20 Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
21 Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: Taint analysis of framework-based web applications. In *OOPSLA'11*, pages 1053–1068. ACM, 2011.
22 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. In *ECOOP'12: Proceedings of the 26th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 435–458, Berlin, Heidelberg, 2012. Springer-Verlag.
23 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. In *ECOOP'12*, LNCS, pages 435–458. Springer-Verlag, 2012.
24 Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE'13*, LNCS, pages 210–225. Springer-Verlag, 2013.
25 Arnaud Venet. Towards the integration of symbolic and numerical static analysis. In *VSTTE 2005*, LNCS, pages 227–236. Springer-Verlag, 2005.
26 Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI'07*, pages 32–41. ACM, 2007.
27 Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *ICSE'08*, pages 171–180. ACM, 2008.
28 Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for javascript. In *ISSTA 2013*, pages 336–346. ACM, 2013.
29 Shiyi Wei and Barbara G. Ryder. State-sensitive points-to analysis for the dynamic behavior of javascript objects. In *ECOOP 2014*, volume 8586 of *LNCS*, pages 1–26. Springer Berlin Heidelberg, 2014.
30 Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06*. USENIX Association, 2006.

Adaptive Context-sensitive Analysis for JavaScript

Shiyi Wei and Barbara G. Ryder

Department of Computer Science
Virginia Tech
Blacksburg, VA, USA
{wei, ryder}@cs.vt.edu

Abstract

Context sensitivity is a technique to improve program analysis precision by distinguishing between function calls. A specific context-sensitive analysis is usually designed to accommodate the programming paradigm of a particular programming language. JavaScript features both the object-oriented and functional programming paradigms. Our empirical study suggests that there is no single context-sensitive analysis that always produces precise results for JavaScript applications. This observation motivated us to design an adaptive analysis, selecting a context-sensitive analysis from multiple choices for each function. Our two-staged adaptive context-sensitive analysis first extracts function characteristics from an inexpensive points-to analysis and then chooses a specialized context-sensitive analysis per function based on the heuristics. The experimental results show that our adaptive analysis achieved more precise results than any single context-sensitive analysis for several JavaScript programs in the benchmarks.

1998 ACM Subject Classification D.3.4 Processors, F.3.2 Semantics of Programming Languages, D.2.5 Testing and Debugging

Keywords and phrases Context Sensitivity, JavaScript, Static Program Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.712

1 Introduction

JavaScript is the *lingua franca* of client-side web applications and is becoming the most popular programming language choice for open source projects [13]. Its flexible programming paradigm is one of the reasons behind its popularity. JavaScript is known as a dynamic object-oriented language, supporting prototype-based programming [9, 20], a different model from class-based languages. However, it also supports features of functional programming (e.g., first class functions). This flexibility helps in programming JavaScript applications for different requirements and goals, but it also renders program analysis techniques more complicated and often ineffective.

Context sensitivity is a general technique to achieve more precise program analysis by distinguishing between calls to a specific function. Historically, call-strings and functional are the two approaches to enable context sensitivity in an analysis [14]. The call-strings approach, using the information on the call stack as a context element[1], originally was explored by Shivers (i.e., known as call-site sensitivity or k-CFA) for functional programming languages [15] and was later adapted to object-oriented languages such as Java [2]. The functional approach, using information about the computation state as a context element, was investigated in several variations for object-oriented languages (e.g., [1, 10]). Several

[1] A *context element* refers to the label that is used to distinguish between different function calling contexts.

© Shiyi Wei and Barbara G. Ryder;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 712–734

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

studies were performed to compare the precision of different context-sensitive analyses for Java [7, 16]. These studies revealed that object sensitivity [10], an functional approach that uses the receiver object represented by its allocation site as a context element, often produced more precise results than call-site sensitivity in Java applications.

Despite the challenges of effective analysis for JavaScript programs, different context-sensitive analyses have been developed to improve precision. Jensen *et al.* implemented object sensitivity in the TAJS analysis framework [4]. Several other context-sensitive analyses were designed for specific features in JavaScript programs (e.g., dynamic property accesses [19] and object property changes [22]). These approaches have been demonstrated effective when certain program features are present. Recently, Kashyap *et al.* conducted a study to compare the performance and precision of different context-sensitive analyses for JavaScript [5]. Unlike the results for Java analyses [7, 16], the authors concluded that there was no clear winner context-sensitive analysis for JavaScript across all benchmarks [5].

Because JavaScript features flexible programming paradigms and no single context-sensitive analysis seems best for analyzing JavaScript programs, there are opportunities for an adaptive (i.e., multi-choice) analysis to improve precision. In this work, we first performed a fine-grained study that compares the precision of four different JavaScript analyses on the function level (Section 2.2) on the same benchmarks used by Kashyap *et al.* [5]. We observed that JavaScript functions in the same program may benefit from use of different context-sensitive analyses depending on specific characteristics of the functions.

The results of our empirical study guided us to design a novel adaptive context-sensitive analysis for JavaScript that selectively applies a specialized context-sensitive analysis per function chosen from call-site, object and parameter sensitivity. This two-staged analysis first applied an inexpensive points-to analysis (i.e., mostly context-insensitive, Section 2.2) to a JavaScript program to extract function characteristics. Each function characteristic is relevant to the precision of a context-sensitive analysis of the function (e.g., the call sites that invoke function *foo* determine the context elements to be generated if call-site sensitivity is applied to *foo*). We designed heuristics according to our observations on the relationship between function characteristics and the precision of a specific analysis using the empirical results on the benchmark programs. Finally, an adaptive analysis based on the heuristics-based selection of a context-sensitive analysis per function was performed. We have evaluated our new approach on two sets of benchmarks. The experimental results show that our adaptive context-sensitive analysis produces more precise results than any single context-sensitive analysis we evaluated for several JavaScript programs and that the heuristics for selecting a context-sensitive analysis per function are fairly accurate.

The major contributions of this work are:

- *A new empirical study on JavaScript benchmarks to compare the precision of different context-sensitive analyses* (i.e., 1-call-site, 1-object and 1st-parameter as defined in Section 2.1). We measure the precision *per function* on two simple clients of points-to analysis (i.e., *Pts-Size* and *REF*). We have made several observations: (i) any specific context-sensitive analysis is precise only for a subset of programs in the benchmarks. More interestingly, any specific context-sensitive analysis is often effective only on a portion of a program. (ii) The precision of a context-sensitive analysis also is dependent on the analysis client. These findings motivated and guided us to design a novel JavaScript analysis.
- *An adaptive context-sensitive analysis for JavaScript.* The heuristics to select from various context-sensitive analyses (i.e., 1-call-site, 1-object and ith-parameter) for a function are based on the function characteristics computed from an inexpensive points-to solution. This adaptive analysis is the *first* analysis for JavaScript that automatically and selectively

applies a context-sensitive analysis depending on the programming paradigms (i.e., coding styles) of a function.

- *An empirical evaluation of the adaptive context-sensitive analysis on two sets of benchmark programs.* The experimental results show that our adaptive analysis was more precise than any single context-sensitive analysis for several applications in the benchmarks, especially for those using multiple programming paradigms. Our results also show that the heuristics were accurate for selecting appropriate context-sensitive analysis on the function level.

Overview. Section 2 introduces various context-sensitive analyses and then presents our motivating empirical study. Section 3 discusses heuristics that represent the relationship between function characteristics and analysis precision. Section 4 describes the design of our new analysis. Section 5 presents experimental results. Section 6 further discusses related work. Section 7 offers conclusions.

2 Background and Empirical Study

In this section, we first introduce the context-sensitive analyses used in our study. We then present the new comparisons among different context-sensitive analyses with empirical results and observations.

2.1 Context Sensitivity

As discussed in Section 1, various context-sensitive analyses have been designed for different programming languages. In this section, we only discuss the approaches that are most relevant to our empirical study of context-sensitive analysis for JavaScript. We provide more discussions on related context-sensitive analyses in Section 6.

Call-strings approach. A call-strings approach distinguishes function calls using information on the call stack. The most widely known call-strings approach is call-site-sensitive (k-CFA) analysis [15]. A k-call-site sensitive analysis uses a sequence of the top k call sites on the call stack as the context element. k is a parameter that determines the maximum length of the call string maintained to adjust the precision and performance of call-site-sensitive analysis. We used *1-call-site sensitivity* for comparison. 1-call-site-sensitive analysis separately analyzes each different call site of a function. Intuitively in the code example below, 1-call-site-sensitive analysis will analyze function *foo* in two calling contexts *L1* and *L2*, such that local variables (including parameters) of *foo* will be analyzed independently for each context element.

```
L1: x.foo(p1, p3);
L2: y.foo(p2, p4);
```

Functional approach. A functional approach distinguishes function calls using information about the computation state at the call. Object sensitivity analyzes a function separately for each of the abstract object names on which this function may be invoked [10]. Milanova *et al.* presented object sensitivity as a parameterized k-object-sensitive analysis, where k denotes the maximum sequence of allocation sites to represent an object name. We used *1-object sensitivity* for comparison. 1-object-sensitive analysis separately analyzes a function for each of its receiver objects with a different allocation site. Intuitively in the code example above, 1-object-sensitive analysis will analyze function *foo* separately if x and/or y may point to

different abstract objects. If x points to objects O_1 and O_2, while y points to object O_3, 1-object-sensitive analysis will analyze function *foo* for three context elements differentiated as O_1, O_2 and O_3.

Other functional approaches presented use the computation state of the parameter instead of the receiver object as a context element [1, 19]. The Cartesian Product Algorithm (CPA) uses tuples of parameter types as a context element for Self [1]. The context-sensitive analysis presented by Sridharan *et al.*, designed specifically for JavaScript programs, analyzes a function separately using the values of a parameter p if p is used as the property name in a dynamic property access (e.g., *v[p]*) [19]. To capture these approaches, we define a simplified, parameterized ith-parameter-sensitive analysis, where i means we use the abstract objects corresponding to the *ith* parameter as a context element. We used *1st-parameter sensitivity* for comparisons. Intuitively in the code example above, 1st-parameter-sensitive analysis will analyze function *foo* separately if *p1* and/or *p2* may point to different abstract objects. If *p1* points to object O_4, while *p2* points to object O_4 and O_5, 1st-parameter-sensitive analysis will analyze function *foo* for two context elements distinguished as O_4 and O_5.

2.2 Empirical Study

There is no theoretical comparison between the functional and call-strings approaches to context sensitivity that proves one better than the other. Therefore, we study the precision of different context-sensitive analyses in practice based on experimental observations. Such comparisons have been conducted for call-site and object sensitivity on Java [7, 16] as well as JavaScript [5] applications. Object sensitivity produced more precise results for Java benchmarks [7, 16], while there was no clear winner across all benchmarks for JavaScript [5]. The latter observation motivated us to perform an in-depth, fine-grained, function-level study which led to our design of a new context-sensitive analysis for JavaScript.

Hypothesis. Our hypothesis is that a specific context-sensitive analysis may be more effective on a portion of a JavaScript program (i.e., some functions), while another kind of context sensitivity may produce better results for other portions of the same program. To test this hypothesis, we compared the precision of different context-sensitive analyses at the function level (i.e., we collect the results of different analyses for a specific function and compare their precision). In contrast, all previous work [7, 16, 5] reported overall precision results per benchmark program. The results of our empirical study were obtained on a 2.4 GHz Intel Core i5 MacBook Pro with 8GB memory running the Mac OS X 10.10 operating system.

Analyses for comparisons. We compared across four different flow-insensitive analyses to study their precision. The *baseline* analysis is an analysis that applies the default context-sensitive analysis for JavaScript in *WALA*[2] (i.e., only uses 1-call-site-sensitive analysis for the constructors to name abstract objects by their allocation sites and for nested functions to properly access variables accessible through lexical scoping [19]). In the implementation, the other three analyses (i.e., 1-call-site, 1-object and 1st-parameter) all apply this default analysis. In principle, these analyses should be at least as precise as the baseline analysis for all functions.

[2] Our implementation is based on the IBM T.J. Watson Libraries for Analysis (*WALA*). `http://wala.sourceforge.net/`

Analysis clients and precision metrics. We compare precision results on two simple clients of points-to analysis. Points-to analysis calculates the set of values a reference property or variable may have during execution, an important enabling analysis for many clients. The first client, *Pts-Size*, is a points-to query returning the cardinality of the set of all values of all local variables in a function (i.e., the total number of abstract objects pointed to by local variables). The second client, *REF* [22], is a points-to query returning the cardinality of the set of all property values in all the property read (e.g., x=$y.p$) or call statements (e.g., $x = y.p(...)$) in a function (i.e., the total number of abstract objects returned by all the property lookups). Because both clients count the number of abstract objects returned and object-naming scheme is unified across different analyses, if an analysis A_1 produces a smaller result than another analysis A_2 for a function *foo*, we say that A_1 is more precise than A_2 for *foo*.

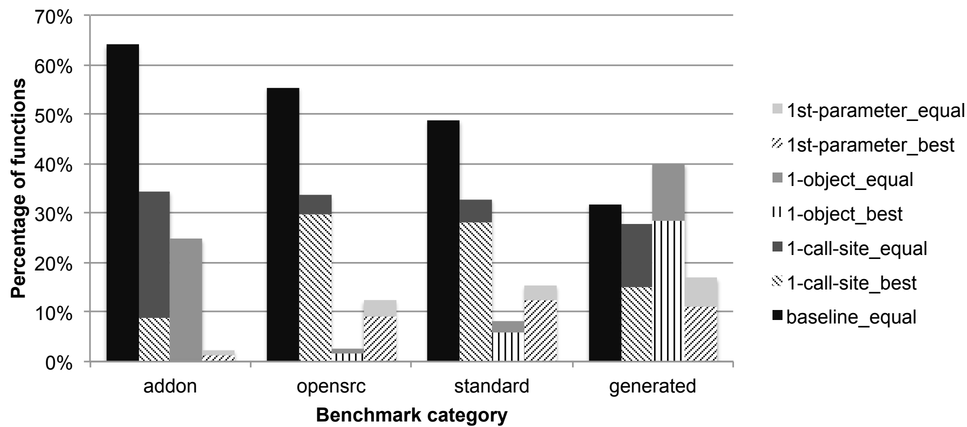
Benchmarks. We conduct our comparisons on the same set of benchmarks used for the study performed by Kashyap *et al.* [5]. There are in total 28 JavaScript programs divided into four categories: *standard* (i.e., from SunSpider[3] and V8[4]), *addon* (i.e., Firefox browser plugins), *generated* (i.e., from the Emscripten LLVM test suite[5]) and *opensrc* (i.e., open source JavaScript frameworks). There are seven programs in each benchmark category.[6] In our study, all benchmark programs are transformed with function extraction for correlated property accesses [19] before running any analysis. This program transformation preserves the semantics of the original programs and may help handle dynamic property accesses more accurately [19].

Results. We ran each analysis of a benchmark program under a time limit of 10 minutes. The baseline and 1-call-site-sensitive analyses finished analyzing all 28 programs under the time limit. 1-object-sensitive analysis timed out on 4 programs (i.e., *linq_aggregate*, *linq_enumerable* and *linq_functional* in the *opensrc* benchmarks and *fourinarow* in the *generated* benchmarks), while 1st-parameter-sensitive analysis timed out on 2 programs (i.e., *fasta* and *fourinarow* in the *generated* benchmarks).

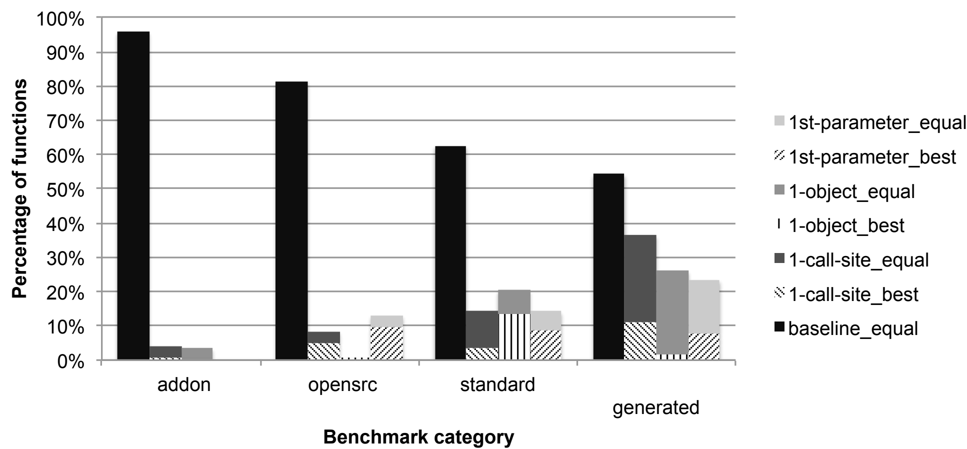
Figures 1a and 1b show the relative precision results for *Pts-Size* and *REF*, respectively. In both figures, the horizontal axis represents the results from four benchmark categories (i.e., *standard*, *addon*, *generated* and *opensrc*) and the vertical axis represents the percentages of functions in each benchmark category on which an analysis produces the *best* results (i.e., more precise results than those from all other three analyses) or *equally precise* results. We consider the results of an analysis as *equally precise* as follows. (i) Baseline analysis is *equally precise* on a function if its results are as precise as each of the other three context-sensitive analyses. (ii) 1-call-site-sensitive, 1-object-sensitive or 1st-parameter-sensitive analysis is *equally precise* on a function if the results are more precise results than baseline analysis, and if the analysis does not produce the *best* results but the results are at least as precise as the other two context-sensitive analyses. This definition indicates that multiple context-sensitive analyses (e.g., 1-call-site-sensitive and 1-object-sensitive analyses) may produce *equally precise* results on a function.

For example, the left four bars in Figure 1a present the precision results for the *addon*

[3] `https://www.webkit.org/perf/sunspider/sunspider.html`
[4] `http://v8.googlecode.com/svn/data/benchmarks/v7/run.html`
[5] `http://kripken.github.io/emscripten-site/`
[6] Details of the benchmark programs were provided in Kashyap *et al.* [5].



(a) Relative precision results for the *Pts-Size* client.



(b) Relative precision results for the *REF* client.

Figure 1 Comparison results between baseline, 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses. Bars show the percentage of functions on which an analysis is *best* or *equally precise*.

benchmarks for the *Pts-Size* client. The *baseline_equal* bar (i.e., the left most) shows that baseline analysis achieved as precise results as those from all three other analyses for 64% of the functions in the *addon* benchmarks, indicating context sensitivity does not make much difference for more than two thirds of the functions in these programs for the *Pts-Size* client. The *1-call-site_best* and *1st-parameter_best* bars (i.e., the parts of the second and fourth bars from left filled with patterns) show that 1-call-site-sensitive and 1st-parameter-sensitive analyses produced more precise results than all other analyses for 9% and 1.5% of the functions in the *addon* benchmarks, respectively. The *1-object_best* result missing from the third bar from left indicates that 1-object-sensitive analysis failed to produce the most precise results for any function in *addon* benchmarks. Nevertheless, the *1-object_equal* bar shows 1-object-sensitive analysis achieved *equally precise* results with 1-call-site-sensitive and/or 1st-parameter-sensitive analyses for 25% of the functions in the *addon* benchmarks.

Comparing with the *1-call-site_equal* (26%) and *1st-parameter_equal* (1%) bars, we can predict that 1-call-site-sensitive and 1-object-sensitive analyses had similar precision on a quarter of the functions in the *addon* benchmarks.

The *baseline_equal* bars in Figure 1a show that analysis of a large percentage of functions in the benchmarks does not benefit from context sensitivity in terms of the *Pts-Size* results (i.e., from 32% for the *generated* benchmarks to *64%* for the *addon* benchmarks). Also, 1-call-site-sensitive analysis had relatively consistent impact in the benchmarks, achieving *best* or *equally precise* results for around 30% functions across all benchmark categories. In contrast, the precision of 1-object-sensitive analysis results seems dependent on the benchmark. Having little impact on the precision of the *opensrc* benchmarks, 1-object-sensitive analysis produced *best* results for 29% of the functions in the *generated* benchmarks with an additional 11% of the functions achieving *equally best* results. 1st-parameter-sensitive analysis, less studied in previous work, produced *best* results for about 10% functions in the *opensrc*, *standard* and *generated* benchmarks, a reasonable technique to improve precision for these programs. It is also interesting to learn from Figure 1a that different context-sensitive analyses may produce *equally precise* results on many functions in some benchmark categories. 1-call-site-sensitive and 1-object-sensitive analyses produced *equally best* results for 25% functions in the *addon* benchmarks. 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses produced *equally best* results for 3% functions in the *generated* benchmarks.

Recall that the *REF* client uses data from different parts of the points-to results than the *Pts-Size* client; in addition, the *REF* client may query the points-to results (i.e., all the property lookup statements in a function) less frequently than the *Pts-Size* client (i.e., all local variables in a function). Overall in Figure 1b, context sensitivity improves precision less over baseline analysis for the *REF* than for the *Pts-Size* client. Baseline analysis produced as precise results as all other three analysis for more than 50% of the functions in all benchmark categories. About 96% of the functions in the *addon* benchmarks did not benefit from any context-sensitive analysis over the baseline analysis. 1-call-site-sensitive analysis achieved dramatically better results for the *Pts-Size* client than *REF* client in *addon*, *opensrc* and *standard* benchmarks. 1-object-sensitive analysis also achieved much better results for the *Pts-Size* client than the *REF* client in the *generated* benchmarks. On the other hand, 1st-parameter-sensitive analysis still remains effective in the *opensrc*, *standard* and *generated* benchmarks.

Summary. First, the effectiveness of specific context-sensitive analysis for JavaScript functions is sensitive to the coding style. For example, 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-senstive analyses each produced *best* results on a large percentage of functions in the programs from the *generated* benchmarks. Second, the precision of context-sensitive analysis also depends on the analysis client.

Based on these observations, we believe JavaScript programs can benefit from an adaptive context-sensitive analysis that chooses an appropriate context-sensitive analysis for a specific function. We used these observations as guidance to design our new analysis.

3 Function Characteristics and Heuristics

A context-sensitive analysis is designed to be useful for a specific programming paradigm. For example, object-sensitive analysis targets the class-based model of object-oriented languages. The results from Section 2 indicate that JavaScript functions in one program may benefit from different context-sensitive analyses depending on the coding style of the functions. In

Table 1 Function characteristics.

	1-call-site	**1-object**	**1st-parameter**
context element approximations	*FC1: CSNum*	*FC4: RCNum*	*FC6: 1ParNum*
	FC2: EquivCSNum		
client-related metrics	*FC3: AllUse*	*FC5: ThisUse*	*FC7: 1ParName*
			FC8: 1ParOther

this section, we present the function characteristics we extracted from the baseline analysis results and then investigate heuristics that represent the relations between these function characteristics and the precision of context-sensitive analyses. Finally, we use these heuristics to select the appropriate context-sensitive analysis per function in our adaptive algorithm (Section 4).

3.1 Function Characteristics

For a JavaScript function, we extracted characteristics from the baseline analysis results that are relevant to the precision of context-sensitive analyses for a specific client. The goal is to extract function characteristics that (i) intuitively are relevant to the precision of a specific analysis for a particular client, and (ii) do not require more costly analysis than a baseline points-to analysis. Table 1 shows that for a JavaScript function *foo*, we extract eight function characteristics (i.e., *FC1*, *FC2*, ..., *FC8*). Each function characteristic is related to the precision of a specific context-sensitive analysis (i.e., *FC1-FC3*, *FC4-FC5*, and *FC6-FC8* are related to the precision of 1-call-site, 1-object and 1st-parameter, respectively).

For a specific context-sensitive analysis, we extracted two kinds of function characteristics: *context element approximations* and *client-related metrics*. A context element approximation predicts the number of distinct context elements generated for a function by a context-sensitive analysis, which determines its ability to distinguish between function calls. A client-related metric predicts the effectiveness of a context-sensitive analysis on a JavaScript function for a particular client. *FC1-FC2*, *FC4* and *FC6* in Table 1 are the context element approximations we designed for 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses, respectively. For example, *FC4-RCNum* presents an approximation of the number of receiver objects on which a function is called, computed from the baseline points-to results. *FC3*, *FC5* and *FC7-FC8* are the client-related metrics we designed for the *Pts-Size* client[7] for 1-call-site, 1-object and 1st-parameter sensitivity, respectively. For example, *FC5-ThisUse* measures the usage frequency of the *this* object in the function body, because frequent use of the *this* object indicates that the precision of the *Pts-Size* client on the function depends on how accurately the *this* object is analyzed. Because 1-object-sensitive analysis potentially analyzes the *this* object more precisely, we use *FC5-ThisUse* to predict the effectiveness of 1-object-sensitive analysis on a function for the *Pts-Size* client. We now will define each function characteristic.

[7] In this work, we use the *Pts-Size* client to demonstrate the effectiveness of our approach because the empirical results in Section 2.2 suggest that the *Pts-Size* client is relatively more sensitive to the choice of context-sensitive analyses.

Function characteristics for 1-call-site-sensitive analysis. Recall that a 1-call-site-sensitive analysis uses the immediate call site of a function as the context element. We define *FC1*, the *CSNum* metric, as follows:

- *FC1-CSNum*: for function *foo*, the number of call sites that invoke *foo* in the baseline call graph *G*.

Although *FC1* approximates the number of context elements that a 1-call-site-sensitive analysis would generate for *foo*, this metric may not be directly relevant to the precision of 1-call-site-sensitive analysis. Intuitively, if function *foo* is invoked from two call sites *CS1* and *CS2*, the analysis precision on *foo* is not likely to benefit from distinguishing between these two call sites if the parameters of *CS1* and *CS2* have the same values because these parameters are used in *foo* as local variables. More precisely, we define two call sites, *CS1: p0.foo(p1, p2, ... , pn)* and *CS2: p0'.foo(p1', p2', ..., pn')*, to be *equivalent* if for each pair of receiver objects and parameters (i.e., *pi* and *pi'*) in *CS1* and *CS2*, the points-to sets of *pi* and *pi'* are the same. We then define *FC2*, the *EquivCSNum* metric using this definition of equivalent call sites, as follows:

- *FC2-EquivCSNum*: for function *foo*, the number of equivalence classes of call sites that invoke *foo* in the baseline call graph *G*.

Recall that the *Pts-Size* client calculates the cardinality of the set of abstract objects to which a local variable of *foo* may point. Intuitively, the precision of the *Pts-Size* client depends on the receiver object or parameters that are frequently used as local variables in the function body. For example, if a parameter *p* is never used in *foo*, even if 1-call-site-sensitive analysis distinguishes call sites that pass different values of *p*, the results of the *Pts-Size* client may not be different because *p* is never used locally. Theoretically, 1-call-site sensitivity may distinguish objects passed through any parameter as well as receiver objects via call sites. We define *FC3*, the *AllUse* metric, as follows:

- *FC3-AllUse*: for function *foo*, the total number of uses of the *this* object and all parameters.

Function characteristics for 1-object-sensitive analysis. Recall that 1-object-sensitive analysis distinguishes calls to a function if they correspond to different receiver objects. To approximate the number of context elements generated by 1-object-sensitive analysis for function *foo*, we define *FC4*, the *RCNum* metric, as follows:

- *FC4-RCNum*: for function *foo*, the total number of abstract receiver objects from all call sites that invoke *foo* in the baseline call graph *G*.

Naturally, 1-object-sensitive analysis would be effective on functions implemented with the object-oriented programming paradigm. The behavior of these functions is dependent on the objects on which they are called. Uses of the *this* object in a function is common in the object-oriented programming paradigm and 1-object-sensitive analysis should produce relatively precise results. We define *FC5*, the *ThisUse* metric, as follows:

- *FC5-ThisUse*: for function *foo*, the total number of uses of the *this* object.

Function characteristics for 1st-parameter-sensitive analysis. The ith-parameter sensitivity is designed to be effective when a specific parameter (e.g., the first parameter for 1st-parameter-sensitive analysis) has large impact on analysis precision. 1st-parameter-sensitive analysis uses the objects that the 1st parameter points to as context elements. We define *FC6*, the *1ParNum* metric, as follows:

- *FC6-1ParNum*: for function *foo*, the total number of abstract objects to which the 1st parameter may point from all call sites that invoke *foo* in the baseline call graph *G*.

If a parameter p is frequently used in a function, it may be more important to apply context-sensitive analysis on p than on the receiver object. Also, if p is used as a property name in dynamic property accesses, using context sensitivity to distinguish the values of p significantly improves analysis precision [19]. We define *FC7*, the *1ParName* metric, and *FC8*, the *1ParOther* metric, as follows:

- *FC7-1ParName*: for function *foo*, the total number of uses of the 1st parameter as a property name in dynamic property accesses.
- *FC8-1ParOther.* for function *foo*, the total number of uses of the 1st parameter not as a property name.

3.2 Heuristics

The function characteristics defined in Section 3.1 are intuitive and easy to calculate from the baseline points-to graph and call graph. Nevertheless, it is still not clear how these function characteristics are related to the precision of a context-sensitive analysis. In this section, we use empirical data to design the heuristics that define the relations between function characteristics and analysis precision.

Our goal is to select an appropriate analysis for a function given the set of its function characteristics. The heuristics are not obvious given that there are multiple context-sensitive analysis choices. To design useful heuristics, we first compared the precision of a pair of analyses on the function level and observed the impact of a subset of function characteristics on these two analyses. We then applied these heuristics to adaptively choose an appropriate context-sensitive analysis using the function characteristics (Section 4). More specifically, for the *Pts-Size* results from the benchmarks (Section 2.2), we compared the precision between all 2-combinations of baseline, 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses and derived the heuristics to select an analysis from each of the combinations.

For example, to choose between the baseline analysis and 1-call-site-sensitive analysis, we obtained the relevant subset of function characteristics (i.e., *FC1-FC3*) and the *Pts-Size* results of baseline and 1-call-site-sensitive analyses for each function *foo* in the benchmarks. If the *Pts-Size* result from 1-call-site-sensitive analysis is more precise than baseline analysis, 1-call-site sensitivity should be chosen when analyzing *foo*; otherwise, baseline analysis should be chosen. Given the list of function characteristics and corresponding analysis choices on the benchmark functions, we first used a machine learning algorithm[8] to get the relationship (i.e., presented as a decision tree) between the function characteristics and analysis choice. We then manually adjusted the initial decision tree based on domain knowledge to decide on the heuristic, in order to ensure that the heuristic is intuitive and easy to interpret while the classifications still maintain good accuracy.

We report the accuracy of an analysis choice (e.g., 1-call-site-sensitive analysis) using the standard information retrieval metrics of *precision* and *recall*. Assuming *S1* is the set of all functions in the benchmarks where 1-call-site-sensitive analysis produces more precise results than baseline analysis and *S2* is the set of functions where 1-call-site-sensitive analysis is chosen by the heuristic. The precision of 1-call-site sensitivity classification is computed as $P_{1-call-site} = \frac{|S1 \bigcap S2|}{|S2|}$ and the recall is computed as $R_{1-call-site} = \frac{|S1 \bigcap S2|}{|S1|}$. The *balanced F-score*, the harmonic mean of the precision and recall, is computed as $F_{1-call-site}$

[8] We used the C4.5 classifier [12] implemented in Weka data mining software (`http://www.cs.waikato.ac.nz/ml/weka/`) to derive the initial decision tree.

$= 2 \times \frac{P_{1-call-site} \times R_{1-call-site}}{P_{1-call-site} + R_{1-call-site}}$, where $F_{1-call-site}$ has its best value at 1 and worst score at 0 for choosing 1-call-site-sensitive analysis by this heuristic.

Figure 2 shows the heuristics to make a choice between each pair of the analyses using function characteristic values. We discuss each pair of the analyses in turn:

Baseline vs. 1-call-site. Three function characteristics (i.e., *FC1-FC3*) are relevant to the precision of 1-call-site-sensitive analysis for the *Pts-Size* client. For 1-call-site-sensitive analysis to produce more precise results than baseline analysis, the *prerequisite* is that there is more than one distinct 1-call-site-sensitive context element (i.e., *FC1* $>$ *1*). Figure 2a shows the heuristic to choose between baseline and 1-call-site-sensitive analyses. 1-call-site-sensitive analysis is chosen over baseline analysis for a function *foo* if there is more than one equivalence class of call sites that invoke *foo* (i.e., *FC2* $>$ *1*). This result indicates that the effectiveness of 1-call-site sensitivity depends on its ability to distinguish call sites with different receiver objects or different corresponding parameters. The balanced F-scores for baseline and 1-call-site-sensitive analyses in this heuristic are 0.46 and 0.8, respectively.

Baseline vs. 1-object. *FC4* and *FC5* are relevant to the precision of 1-object-sensitive analysis for the *Pts-Size* client. For 1-object-sensitive analysis to produce more precise results than baseline analysis, the *prerequisite* is that there is more than one 1-object-sensitive context element (i.e., *FC4* $>$ *1*). Figure 2b shows the heuristic to choose between baseline and 1-object-sensitive analyses. 1-object-sensitive analysis is chosen over baseline analysis for a function *foo* if the *this* object is used at least once in the function body of *foo* (i.e., *FC5* $>$ *0*). This result suggests that 1-object-sensitive analysis is useful in terms of *Pts-Size* client for a function *foo* whose behavior relies on the values of the *this* object, even for a small number of 1-object-sensitive context elements for *foo*. The balanced F-scores for baseline and 1-object-sensitive analyses in this heuristic are 0.65 and 0.79, respectively.

Baseline vs. 1st-parameter. Three function characteristics (i.e., *FC6-FC8*) are relevant to the precision of 1st-parameter-sensitive analysis for the *Pts-Size* client. For 1st-parameter-sensitive analysis to produce more precise results than baseline analysis, the *prerequisite* is that there is more than one 1st-parameter-sensitive context element (i.e., *FC6* $>$ *1*). Figure 2c shows the heuristic to choose between baseline and 1st-parameter-sensitive analyses. 1st-parameter-sensitive analysis is chosen over baseline analysis for a function *foo* if the first parameter of *foo* is used (i.e., as the property name in dynamic property accesses or otherwise) at least once in the function body of *foo* (i.e., *FC7* $>$ *0 OR FC8* $>$ *0*). Similar to 1-object sensitivity, 1st-parameter sensitivity is another functional approach that distinguishes calls based on the computation states of a parameter. It is expected for 1-object-sensitive or 1st-parameter-sensitive analysis to be effective on the function *foo* if the values of the *this* object or the first parameter affect the behavior of *foo*. The balanced F-scores for baseline and 1st-parameter-sensitive analyses in this heuristic are 0.49 and 0.83, respectively.

1-call-site vs. 1-object. To select between 1-call-site-sensitive and 1-object-sensitive analyses, function characteristics related to both are considered (i.e., *FC1-FC5*). In our adaptive analysis, two context-sensitive analyses are compared for a function when both of them would be chosen over baseline analysis (see Section 4). As a consequence, the *prerequisite* for the heuristic in this case is the number of equivalence classes of call sites is larger than 1 (i.e., *FC2* $>$ *1*) and the *this* object is used at least once (i.e., *FC5* $>$ *0*). Figure 2d shows the heuristic to choose between 1-call-site-sensitive and 1-object-sensitive analyses. The heuristic

```
FC2-EquivCSNum = 1: baseline
FC2-EquivCSNum > 1: 1-call-site
```

(a) Baseline vs. 1-call-site.

```
FC5-ThisUse = 0: baseline
FC5-ThisUse > 0: 1-object
```

(b) Baseline vs. 1-object.

```
FC7-1ParName = 0 AND FC8-1ParOther = 0: baseline
FC7-1ParName > 0 OR FC8-1ParOther > 0: 1st-parameter
```

(c) Baseline vs. 1st-parameter.

```
FC4-RCNum / FC2-EquivCSNum <= 0.8: 1-call-site
FC4-RCNum / FC2-EquivCSNum > 0.8
|   FC5-ThisUse / FC3-AllUse <= 0.375: 1-call-site
|   FC5-ThisUse / FC3-AllUse > 0.375: 1-object
```

(d) 1-call-site vs. 1-object.

```
FC7-1ParName = 0
|   FC8-1ParOther / FC3-AllUse <= 0.19: 1-call-site
|   FC8-1ParOther / FC3-AllUse > 0.19
|   |   FC6-1ParNum / FC2-EquivCSNum <= 3.8
|   |   |   FC8-1ParOther / FC3-AllUse <= 0.35: 1-call-site
|   |   |   FC8-1ParOther / FC3-AllUse > 0.35: 1st-parameter
|   |   FC6-1ParNum / FC2-EquivCSNum > 3.8: 1st-parameter
FC7-1ParName > 0: 1st-parameter
```

(e) 1-call-site vs. 1st-parameter.

```
FC7-1ParName = 0
|   FC5-ThisUse / FC8-1ParOther <= 0.8: 1st-parameter
|   FC5-ThisUse / FC8-1ParOther > 0.8
|   |   FC5-ThisUse / FC8-1ParOther <= 1.34
|   |   |   FC4-RCNum / FC6-1ParNum < 0.5: 1st-parameter
|   |   |   FC4-RCNum / FC6-1ParNum >= 0.5
|   |   |   |   FC4-RCNum / FC6-1ParNum <= 1: unknown
|   |   |   |   FC4-RCNum / FC6-1ParNum > 1: 1-object
|   |   FC5-ThisUse / FC8-1ParOther > 1.34: 1-object
FC7-1ParName > 0: 1st-parameter
```

(f) 1-object vs. 1st-parameter.

Figure 2 Heuristics to select between a pair of analyses.

consists of the relationship between the metrics of both analyses. 1-call-site-sensitive analysis is selected if it generates a greater number of context elements than 1-object-sensitive analysis (i.e., *FC4 / FC2 <= 0.8*) for a function. This result suggests that 1-call-site-sensitive and 1-object-sensitive analyses in this case are empirically comparable in terms of precision. The relationship between the numbers of context elements generated by each analysis on *foo* indicates which context-sensitive analysis may be more precise for that function. When the

number of receiver objects that invoke *foo* is close to or larger than the number of equivalence classes of call sites (i.e., $FC4 / FC2 > 0.8$), if the *this* object is used quite frequently (i.e., $FC5 / FC3 > 0.375$) in *foo*, 1-object-sensitive analysis is more precise for *foo*; otherwise (i.e., $FC5 / FC3 <= 0.375$), 1-call-site-sensitive analysis is selected. This result is intuitive in that 1-object-sensitive analysis produces more precise results than 1-call-site-sensitive analysis for the *Pts-Size* client when (i) 1-object-sensitive analysis generates a number of context elements and (ii) the behavior of a function is heavily dependent on the values of the receiver object. The balanced F-scores for 1-call-site-sensitive and 1-object-sensitive analyses in this heuristic are 0.67 and 0.8, respectively.

1-call-site vs. 1st-parameter. Function characteristics *FC1-FC3* and *FC6-FC8* are considered to select between 1-call-site-sensitive and 1-object-sensitive analyses. The *prerequisite* for this comparison is $FC2 > 1$ and the first parameter of the function is used at least once (i.e., $FC7 > 0$ *OR* $FC8 > 0$). Figure 2e shows the heuristic to choose between 1-call-site-sensitive and 1-object-sensitive analyses. 1st-parameter-sensitive analysis is always selected if the first parameter is ever used as a property name in dynamic property accesses because the dynamic property accesses in JavaScript make analysis results very imprecise [19] and 1st-parameter sensitivity is a technique that significantly improves the analysis precision in this situation. In other cases, 1-call-site sensitive analysis is preferred if uses of the first parameter are not important to the function behavior (i.e., $FC8 / FC3 <= 0.19$). Also, similar to the heuristic that selects between 1-call-site-sensitive and 1-object-sensitive analyses (Figure 2d), the heuristic between 1-call-site-sensitive and 1st-parameter-sensitive analyses is dependent on the relationship between the context elements generated by both analyses. If 1st-parameter-sensitive analysis potentially generates many more context elements than 1-call-site sensitive analysis (i.e., $FC6 / FC2 > 3.8$), we expect the 1st-parameter-sensitive analysis to be more precise. Otherwise (i.e., $FC6 / FC2 <= 3.8$), depending on the importance of the first parameter to the function behavior, 1-call-site-sensitive analysis (when $0.19 < FC8 / FC3 <= 0.35$) or 1st-parameter-sensitive analysis (when $FC8 / FC3 > 0.35$) is selected. The balanced F-scores for 1-call-site-sensitive and 1st-parameter-sensitive analyses in this heuristic are 0.73 and 0.66, respectively.

1-object vs. 1st-parameter. Finally, Figure 2f presents the heuristic that selects between 1-object-sensitive and 1st-parameter-sensitive analyses. Function characteristics *FC4-FC8* are considered and the *prerequisite* is $FC5 > 0$ and $FC7 > 0$ *OR* $FC8 > 0$. It is not surprising that 1-object-sensitive analysis is selected by the heuristic when the *this* object is more frequently used (i.e., $FC5 /FC8 > 1.34$) and 1st-parameter-sensitive analysis is selected when the condition is opposite (i.e., $FC5 /FC8 <= 0.8$). When uses of the *this* object and the first parameter are similar (i.e., $0.8 < FC5 /FC8 < 1.34$), the number of context elements generated by these two analyses decides the selection: (i) if 1-object-sensitive analysis potentially generates more context elements than 1st-parameter-sensitive analysis (i.e., $FC5 /FC\ 8 > 1$), we expect 1-object sensitive analysis to be more precise; (ii) if 1st-parameter-sensitive analysis generates more than twice the number of context elements than 1-object-sensitive analysis (i.e., $FC5 /FC\ 8 < 0.5$), 1st-parameter-sensitive analysis is selected; (iii) otherwise (i.e., $0.5 <= FC5 /FC\ 8 <= 1$), it is not clear from the data in the benchmarks which analysis produces more precise results because the function characteristics indicate that they have similar capability to analyze the function. In this case, we randomly select between 1-object-sensitive and 1st-parameter-sensitive analyses for the function whose characteristics fall in this region. The balanced F-scores for 1-object-sensitive and 1st-parameter-sensitive analyses in this heuristic are 0.79 and 0.86, respectively.

```
iParName < jParName: jth-parameter
iParName = jParName
|   iParOther < jParOther: jth-parameter
|   iParOther = jParOther
|   |   iParNum < jParNum: jth-parameter
|   |   iParNum >= jParNum: ith-parameter
|   iParOther > jParOther: ith-parameter
iParName > jParName: ith-parameter
```

Figure 3 Heuristic for ith-parameter vs. jth-parameter.

Summary. The heuristics presented in Figure 2 are intuitive for making a choice between each pair of analyses. More importantly, the heuristics for the call-strings approach and the functional approaches (i.e., Figures 2d and 2e) allow us to make a decision between two incomparable analyses. Finally, these heuristics are accurate (i.e., good balanced F-scores) in terms of their effectiveness on the benchmark programs.

4 Adaptive Context-sensitive Analysis

In this section, we present our adaptive context-sensitive analysis algorithm. This staged analysis (i) uses baseline analysis to obtain a call graph and points-to solution to extract function characteristics and (ii) performs an adaptive context-sensitive analysis based on heuristics that select an appropriate context-sensitive analysis for each function.

4.1 Function Characteristics Extraction

In Section 3, we discussed the function characteristics used in the heuristics to select from baseline, 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses. Various other context-sensitive analyses exist for improving analysis precision. For example, ith-parameter-sensitive analysis (Section 2.1) provides variations to distinguish function calls based on the computation states of parameters, decided by the parameter i.

In our adaptive context-sensitive analysis, we actually apply ith-parameter-sensitive analysis for a function whose precision relies on how accurately the ith parameter is analyzed. Therefore, for each parameter of a function, we extract three function characteristics: *iParNum*, *iParName* and *iParOther*. We apply the heuristics of 1st-parameter-sensitive analysis to select between ith-parameter-sensitive analysis and baseline(Figure 2c), 1-call-site-sensitive (Figure 2e) or 1-object-sensitive (Figure 2f) analysis. To select between ith-parameter-sensitive and jth-parameter-sensitive analyses, we apply the heuristic shown in Figure 3. We designed this heuristic based on the observation that for parameter sensitivity, a functional approach, the uses of the parameter whose computation states are used to distinguish function calls usually are more closely related to the analysis precision. In Figure 3, because the uses of a parameter as a property name in the dynamic property accesses is the most important characteristic, if one parameter is used as a property name more often than the other, distinguishing function calls based on its values may produce more precise results. If the *ParName* characteristics are the same for parameters i and j, the uses of the parameters in other situations are compared to decide if ith-parameter-sensitive analysis is more/less precise than jth-parameter-sensitive analysis. Finally, if both client-related metrics (i.e., *ParName* and *ParOther*) cannot distinguish the parameters, the heuristic selects the parameter that points to more objects.

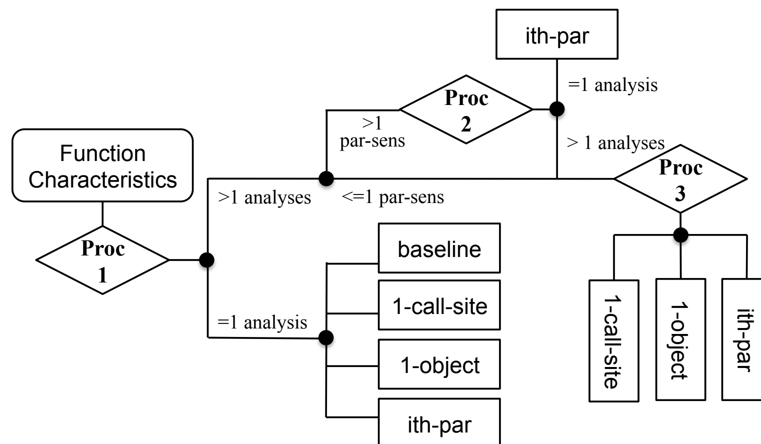


Figure 4 Workflow to select a context-sensitive analysis for a JavaScript function.

For a function *foo* with n parameters, we extract three characteristics for 1-call-site sensitivity (i.e., *CSNum*, *EquivCSNum* and *AllUse*), two characteristics for 1-object sensitivity (i.e., *RCNum* and *ThisUse*), and three characteristics for ith-parameter sensitivity (i.e., *iParNum*, *iParName* and *iParOther*). In total, there are *6+3n* function characteristics for *foo*.

4.2 Algorithm

Our adaptive context-sensitive analysis automatically selects a specific context-sensitive analysis for each function based on the function characteristics derived from the baseline analysis. Figures 2a-2f and 3 present the heuristics used to choose between pairs of analyses. The overall algorithm to select the context-sensitive analysis for a function is described in Figure 4. Given the function characteristics of function *foo*, Procedure 1 performs all pairwise comparisons between baseline analysis and 1-call-site-sensitive, 1-object-sensitive and ith-parameter-sensitive analyses for all the parameters of *foo*. If Procedure 1 returns a single analysis, this analysis is selected for *foo*. Returning baseline analysis means none of the context-sensitive analyses makes much difference to improve precision for *foo*.

In case more than one choice is returned by Procedure 1, further comparisons are conducted to decide the specific context-sensitive analysis to use for *foo*. If the analysis precision of *foo* may benefit from applying parameter-sensitive analyses on multiple parameter choices returned by Procedure 1, Procedure 2 selects from among them to find the parameter i that may produce the most precise results when ith-parameter-sensitive analysis is applied. If the choices from Procedure 1 are now narrowed down to only the ith-parameter-sensitive analysis, this analysis is selected by our algorithm to analyze *foo*.

When necessary, Procedure 3 chooses from the remaining context-sensitive analyses that are returned by Procedures 1 and 2. If there are two remaining context-sensitive analyses to choose from, Procedure 3 applies the heuristic in Figure 2d, 2e or 2f to decide on the context-sensitive analysis for analyzing *foo*. Otherwise (i.e., to choose from all three context-sensitive analyses), Procedure 3 compares each pair of 1-call-site-sensitive, 1-object-sensitive and ith-parameter-sensitive analyses and tries to find a best context-sensitive analysis for a majority of the pairs using heuristics in Figures 2d, 2e and 2f. For example, the adaptive analysis selects 1-call-site-sensitive analysis to analyze *foo* if it is chosen by both heuristics comparisons with 1-object-sensitive and ith-parameter-sensitive analyses. Finally, if Procedure 3 cannot

decide on a specific accurate context-sensitive analysis (i.e, when each of the three heuristics returns a different analysis choice), the adaptive analysis randomly chooses an analysis for *foo*.

5 Evaluation

In this section, we first present the details of our experimental setup. We then evaluate our adaptive context-sensitive analysis using two sets of benchmarks. We compared the precision of adaptive analysis to other context-sensitive analyses applied to the entire program.

5.1 Experiment Setup

Our implementation of adaptive context-sensitive analysis was based on the *WALA* static analysis infrastructure that supports JavaScript analysis. The baseline points-to analysis, *ZERO_ONE_CFA* analysis in *WALA* that uses the default context sensitivity for JavaScript analysis (Section 2.2), produced a call graph and a points-to solution from which we extracted the function characteristics. For the adaptive context-sensitive analysis, we implemented a new context selector[9] that applies the context-sensitive analysis chosen by the heuristics for each function. Note that the default context-sensitive analysis is always used as well to ensure that the results of adaptive analysis are comparable to the baseline analysis.

The goals of the experiments include: (i) comparing the precision of adaptive context-sensitive analysis with each of the other context-sensitive analyses to learn if the adaptive analysis improves JavaScript analysis precision and (ii) studying the accuracy of selecting a specific context-sensitive analysis for each function to validate the quality of the heuristics presented in Section 3.

To achieve these goals, we evaluated our analysis on two sets of benchmarks: (i) the same benchmark programs on which we performed the empirical study in Section 2.2 (i.e., Benchmarks I including the 28 JavaScript programs collected by Kashyap *et al.* [5], divided into four categories) and (ii) four open-source JavaScript applications or libraries (i.e., Benchmarks II). The programs in Benchmarks II are (i) *Box2DWeb*, collected in the *Octane* benchmarks[10], (ii) *minified.js* library[11] version 1.0, (iii) *mootools* library[12] version 1.5.1, and *benchmark.js* library[13] version 1.0.0. Because the heuristics were designed based on machine learning results using Benchmarks I, Benchmarks II serve to test if these heuristics can be applied by the adaptive context-sensitive analysis to arbitrary JavaScript programs and produce fairly accurate analysis results for the *Pts-Size* client.

5.2 Experimental Results

Results for Benchmarks I. Figure 5 shows the analysis precision results for Benchmarks I. We compared the results of our adaptive analysis with the context-sensitive analysis (i.e., 1-call-site-sensitive, 1-object-sensitive or 1st-parameter-sensitive analysis) that produced most accurate results for each program for these benchmarks. We define a context-sensitive analysis to be the *winner* analysis for a program if it was at least as precise as the other

[9] In *WALA*, the context element at a call site is decided by a context selector.
[10] `https://developers.google.com/octane/`
[11] `http://minifiedjs.com`
[12] `http://mootools.net`
[13] `http://benchmarkjs.com`

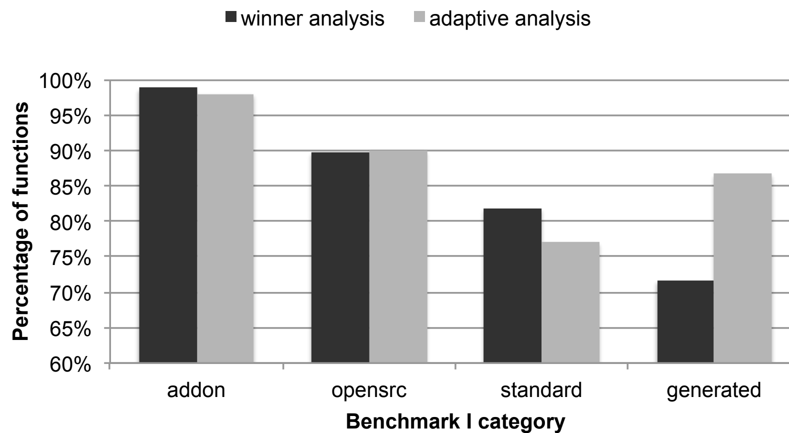


Figure 5 Analysis precision on Benchmarks I.

two context-sensitive analyses on the largest number of functions. For all 14 programs in the *addon* and *opensrc* benchmarks, 1-call-site-sensitive analysis was the *winner* among the 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses. For the *standard* benchmarks, 1st-parameter-sensitive analysis was *winner* on three programs and 1-call-site-sensitive analysis was *winner* on the other four programs. For all but one program in the *generated* benchmarks, 1-object-sensitive analysis was the *winner* analysis and 1-call-site-sensitive analysis was *winner* for *fourinarow*. In Figure 5, the *winner analysis* bar shows the percentage of the total number of functions in each benchmark category on which the *winner* analysis produced most accurate results. For example, the leftmost *winner analysis* bar represents that 1-call-site analysis (i.e., the *winner* analysis for all the programs in the *addon* benchmarks) produced at least as precise results as 1-object-sensitive and 1st-parameter-sensitive analyses for 98.8% of the functions in the *addon* benchmarks. The *adaptive analysis* bar shows the percentage of functions in each benchmark category for which our adaptive analysis produced at least as precise results as 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses.

In Figure 5, our adaptive analysis produced at least as precise results for most functions in the *addon* and *opensrc* benchmarks (i.e., 98% and 90%, respectively). In these two benchmark categories, 1-call-site-sensitive analysis was the *winner* analysis for all programs, producing at least as precise results for 98.8% and 89.8% of the functions, respectively. These results indicate our adaptive analysis is capable of producing similar precision for a set of programs that shares the same *winner* analysis. For the *standard* benchmarks, the *winner* analyses (i.e., 1-call-site-sensitive analysis for four programs and 1st-parameter-sensitive analysis for three programs) were more precise than the adaptive analysis in terms of the percentage of functions for which an analysis produced at least as precise results (i.e., 81.9% of the functions for the *winner* analyses comparing to 77.1% of the functions for adaptive analysis). Nevertheless, the adaptive analysis still achieved good precision for most of these relatively small programs in the *standard* benchmarks. Even though there were different *winner* context-sensitive analyses on the programs from the *standard* benchmarks, the use of the adaptive analysis avoided having to manually pick a specific context-sensitive analysis for an individual program. Finally, for the programs in the *generated* benchmarks, our adaptive analysis significantly improved precision over the *winner* context-sensitive analyses (i.e., 1-object-sensitive analysis for 6 programs and 1-call-site-sensitive analysis for the other one),

Table 2 Selection precision for Benchmarks I.

<table>
<tr><th>best / equally precise analysis</th><th># of observed functions</th><th># of selected functions (true positives)</th><th>true positive rate</th></tr>
<tr><td>1-call-site</td><td>351</td><td>258</td><td>73.5%</td></tr>
<tr><td>1-object</td><td>241</td><td>164</td><td>68.0%</td></tr>
<tr><td>1st-parameter</td><td>162</td><td>79</td><td>48.8%</td></tr>
<tr><td rowspan="2">1-call-site = 1-object</td><td rowspan="2">153</td><td>1-call-site: 39</td><td rowspan="2">94.1%</td></tr>
<tr><td>1-object: 105</td></tr>
<tr><td rowspan="2">1-call-site
= 1st-parameter</td><td rowspan="2">39</td><td>1-call-site: 23</td><td rowspan="2">74.4%</td></tr>
<tr><td>1st-parameter: 6</td></tr>
<tr><td rowspan="2">1-object
= 1st-parameter</td><td rowspan="2">6</td><td>1-object: 4</td><td rowspan="2">83.3%</td></tr>
<tr><td>1st-parameter: 1</td></tr>
<tr><td rowspan="3">1-call-site
= 1-object
= 1st-pararameter</td><td rowspan="3">25</td><td>1-call-site: 2</td><td rowspan="3">100%</td></tr>
<tr><td>1-object: 13</td></tr>
<tr><td>1st-parameter: 10</td></tr>
<tr><td>total</td><td>977</td><td>704</td><td>72.1%</td></tr>
</table>

from 71.7% to 86.7% functions. According to the results in Figure 1a, the programs in the *generated* benchmarks require context sensitivity for precision and moreover, these programs often benefited from different context-sensitive analyses. The results for the *generated* benchmarks show that we achieved our goal to analyze a multi-paradigm JavaScript program more accurately using a different context-sensitive analysis for each function.

The most important aspect of adaptive analysis is its ability to select an appropriate context-sensitive analysis for a specific function. Table 2 shows the accuracy of the analysis selection process for a function using the heuristics presented in Section 3 with the *Pts-Size* client. The first column in Table 2 lists the (set of) analyses that are *best* or *equally precise* (i.e., rows 4-7 in the first column) for a function (see Section 2.2). The second column shows the total number of functions in all programs from Benchmarks I on which the corresponding analyses were observed to produced the *best* or *equally precise* results. There were in total 1817 functions analyzed in Benchmarks I and the precision results of 977 functions were improved over baseline analysis by at least one context-sensitive analysis for the *Pts-Size* client. The last column presents the the number of functions on which the adaptive analysis matched the observed results (i.e., true positives for our heuristics). For those functions on which 1-call-site-sensitive and 1-object-sensitive analyses produced the *best* results, the selection heuristics resulted in good precision (i.e., 73.5% and 68%, respectively). However, the selection on 1st-parameter-sensitive analysis only achieved 48.8% precision. This is because our adaptive analysis chooses the appropriate ith-parameter-sensitive analysis to analyze a function using the parameter sensitivity. Here we are only checking the selection precision with respect to 1st-parameter-sensitive analysis; whereas ith-parameter-sensitive analysis (i>1) was applied to analyze 51 functions in the programs of Benchmarks I.

1-call-site-sensitive and 1-object-sensitive analyses produced *equally precise* results in terms on *Pts-Size* client on 153 functions. The adaptive analysis correctly selected 1-call-site-sensitive or 1-object-sensitive analysis to analyze 144 of those 153 functions, and interestingly, the choice was leaning towards 1-object-sensitive analysis (i.e., 1-object-sensitive analysis for 105 functions comparing to 1-call-site-sensitive analysis for 39 functions). For the functions on which *equally precise* results were produced by 1-call-site-sensitive and

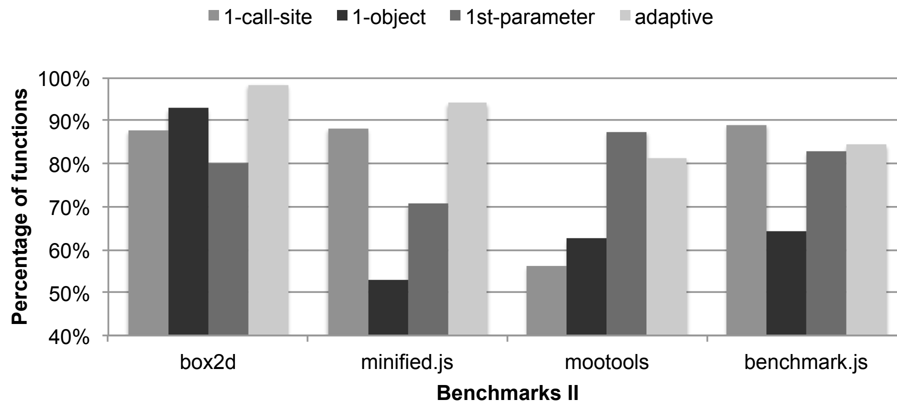


Figure 6 Analysis precision on Benchmarks II.

1st-parameter-sensitive analyses, adaptive analysis selects more functions to be analyzed by 1-call-site-sensitive analysis. The overall precision of selecting a context-sensitive analysis by our heuristics is very good (i.e., 72.1%); this is a measure of when adaptive analysis made the best choice possible. The above observations may help us to improve the heuristics in the future.

The time cost of our adaptive analysis is the sum of its two stages (i.e., the baseline points-to analysis to gather function characteristics and the subsequent adaptive context-sensitive analysis). We compare the performance of our adaptive analysis with the *winner* analysis for each program in Benchmarks I. On average over all the programs in Benchmarks I, our two-staged analysis introduced a 67% overhead. Nevertheless, the second stage (i.e., the adaptive context-sensitive analysis) is on average 19% faster than the *winner* analysis over the Benchmarks I programs. This result suggests that an appropriate choice of context sensitivity per function yields better performance and precision.

Results for Benchmarks II. Figure 6 shows initial analysis precision results using four programs from Benchmarks II. The sizes of these programs, in terms of the number of functions analyzed, are 126, 119, 80 and 64, respectively. The *1-call-site*, *1-object* and *1st-parameter* bars represent the percentage of functions on which each context-sensitive analysis produced at least as precise results as the other two. The *adaptive* bar (rightmost) represents the percentage of functions on which the adaptive analysis produced at least as precise results as 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses. We picked these four programs in Benchmarks II because their analysis results for different context-sensitive analyses were varied. For example, 1-object-sensitive analysis was more precise than 1-call-site-sensitive and 1st-parameter sensitive analyses for *box2d*, while 1st-parameter-sensitive analysis was the most precise for *mootools*.

The results in Figure 6 show that our adaptive analysis achieved better results than any single context-sensitive analysis for *box-2d* and *minified.js*. For example, 1-object-sensitive analysis was at least as precise for 92.8% of the functions in *box-2d*; adaptive analysis improved these results to 98.4% of the functions. 1-call-site-sensitive analysis produced at least precise results for 88.2% of the functions in *minified.js*; adaptive analysis improved the results by 5.9% more functions. 1st-parameter-sensitive and 1-call-site-sensitive analyses were the most precise context-sensitive analyses for *mootools* and *benchmark.js*, respectively. While adaptive analysis produced results lower than these analyses, the results of adaptive analysis

were close, only different for 6.2% and 4.7% of the functions in *mootools* and *benchmark.js*, respectively. Overall, our adaptive context-sensitive analysis was fairly accurate for analyzing these programs from Benchmarks II. This promising result indicates that the heuristics presented in Section 3 may be applicable in general to JavaScript programs.

5.3 Discussion

In this work, we have demonstrated the ability of our adaptive analysis that chooses a specific context-sensitive analysis for each function in order to significantly improve analysis precision. Nevertheless, this initial work has inspired us with more research ideas for further improvements of context-sensitive analyses for JavaScript. First, since we evaluated the adaptive analysis on a simple client of points-to analysis (i.e., *Pts-Size*), it would be interesting to know if adaptive analysis is effective to improve precision for other clients (e.g., security analysis). Second, context-sensitive analysis for JavaScript are not limited to 1-call-site, 1-object and ith-parameter. A deeper object-sensitive analysis (i.e., k-object) or another context-sensitive analysis (e.g., using the length of the parameter list at a call site as context element to distinguish JavaScript variadic functions [21]) could be used by adaptive analysis. New heuristics need to be designed for selecting these analyses. Third, we would like to explore if analysis precision may benefit from applying multiple-sensitive analyses on a specific JavaScript function. The idea of hybrid context-sensitive analysis has been tried for analyzing Java programs [6]. Fourth, scalability is an important issue for JavaScript analyses, especially for analyzing JavaScript websites that use libraries heavily (e.g., jQuery). In this work, we do not address this problem, that is, when a baseline points-to analysis is not scalable for a large JavaScript application, typically a website. In the future, we plan to focus on improving the performance of analysis of JavaScript websites using an adaptive approach.

Although we used benchmarks collected by Kashyap *et al.* [5] as well as other JavaScript programs to evaluate adaptive context-sensitive analysis, these programs may not be representative of non-website JavaScript applications, which might threaten the validity of our conclusions as applicable to all JavaScript programs.

6 Related Work

To the best of our knowledge, we have proposed the first analysis for JavaScript that adaptively uses multiple context-sensitive analyses to analyze a program when context sensitivity may help improve precision. Nevertheless, our work is related to approaches that apply context-sensitive analysis selectively (e.g., refinement-based analysis [3, 18, 17] and hybrid context-sensitive analysis [6]) for other programming languages. We already have discussed several context-sensitive analyses in Sections 1 and 2. In this section, we focus on related "selective" context-sensitive analyses.

Context-sensitive analysis has been deeply investigated for other object-oriented languages such as Java. However, these object-oriented languages do not seem as amenable to our approach of using different context-sensitive analyses on different functions. Castries and Smaragdakis presented hybrid context-sensitive points-to analysis for Java [6]. Several combinations of call-site and object-sensitive analyses were explored and evaluated for precision. Their results showed that selectively adding call-site-sensitive analysis to specific places in the program (e.g., static calls) significantly improved the precision of object-sensitive points-to analysis for Java. Our adaptive analysis automatically chooses an appropriate context-sensitive analysis for each function in JavaScript program.

Several works were proposed to tune the context sensitivity of an analysis based on pre-analysis results. Smaragdakis *et al.* presented introspective analysis that aims to improve the performance of a context-sensitive analysis for Java [17]. Introspective analysis selectively refines allocation sites or call sites based on the heuristics consisting of metrics computed from context-insensitive points-to results. The heuristics are tunable via constant parameters. In our adaptive analysis, the heuristics are computed from baseline analysis and syntactic analysis. The heuristics in our analysis focus on "which" context-sensitive analysis may improve precision instead of "if" context sensitivity would be of benefit.

Sridharan and Bodík presented a refinement-based points-to analysis for Java [18] that refines sensitivity for heap accesses and method calls. It also is demand-driven in that it skips irrelevant code in the analysis. Our adaptive context-sensitive analysis aims to improve precision for the whole program.

Guyer and Lin presented a client-driven analysis for C that automatically adjusts its precision in response to the needs of client analyses [3]. This client-driven analysis monitors polluting assignments (i.e., the program points that result inaccuracy in the analysis) and tunes context as well as flow sensitivity to improve precision. Liang and Naik presented another client-driven algorithm for Java that prunes away analysis results irrelevant to refinement for more precision [8]. For these techniques, a pre-analysis is used to determine the program points for refinement. Baseline points-to analysis is used to derive the function characteristics for our heuristics. Furthermore, our adaptive analysis involves more than one context-sensitive analysis.

Oh et al. presented a selective context-sensitive analysis for C guided by an impact pre-analysis [11]. The impact pre-analysis applies full context sensitivity (i.e., ∞-CFA) but with simplified abstract domain and transfer functions to infer the impacts of context sensitivity in the main analysis. The heuristics in our adaptive analysis focus on the characteristics of a function to indicate whether analysis precision for a function would benefit from a specific context-sensitive analysis. Our pre-analysis, the baseline analysis, is comparable with the adaptive analysis in terms of abstract domain and transfer functions.

7 Conclusions

The effectiveness of a context-sensitive analysis on a JavaScript program depends on its coding style because JavaScript features both object-oriented and functional programming paradigms. The fact that there is no winner context-sensitive analysis for the JavaScript benchmarks we examined motivated us to design an adaptive analysis. Our analysis applies a specialized context-sensitive analysis per function, using heuristics based on function characteristics derived from an inexpensive points-to analysis. Our experimental results show that adaptive analysis is more precise than any single context-sensitive analysis for several programs in the benchmarks, especially for those multi-paradigm programs whose analysis precision can benefit from multiple context-sensitive analyses. This work also has inspired opportunities to solve fundamental problems in analyzing JavaScript programs including analysis scalability for websites.

References

1 Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP'95, pages 2–26, 1995.

2 David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'97, pages 108–124, 1997.

3 Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 214–236, 2003.

4 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS'09, pages 238–255, 2009.

5 Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. Jsai: A static analysis platform for JavaScript. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 121–132, 2014.

6 George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'13, pages 423–434, 2013.

7 Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):3:1–3:53, October 2008.

8 Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'11, pages 590–601, 2011.

9 Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA'86, pages 214–223, 1986.

10 Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, January 2005.

11 Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'14, pages 475–484, 2014.

12 J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

13 RedMonk. The RedMonk programming language rankings. `http://redmonk.com/sogrady/2014/06/13/language-rankings-6-14/`, 2014.

14 Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

15 Olin Grigsby Shivers. *Control-flow Analysis of Higher-order Languages of Taming Lambda*. PhD thesis, Carnegie Mellon University, 1991.

16 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'11, pages 17–30, 2011.

17 Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'14, pages 485–495, 2014.

18 Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'06, pages 387–400, 2006.

19 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 435–458, 2012.

20 Peter Wegner. Dimensions of object-based language design. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA'87, pages 168–182, 1987.

21 Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for JavaScript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 336–346, 2013.

22 Shiyi Wei and Barbara G. Ryder. State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. In *Proceedings of the 28th European Conference on Object-oriented Programming*, ECOOP'14, pages 1–26, 2014.

Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity

Changhee Park and Sukyoung Ryu

Department of Computer Science, KAIST, Daejeon, Republic of Korea
{changhee.park,sryu.cs}@kaist.ac.kr

Abstract

The numbers and sizes of JavaScript applications are ever growing but static analysis techniques for analyzing large-scale JavaScript applications are not yet ready in a *scalable* and *precise* manner. Even when building complex software like compilers and operating systems in JavaScript, developers do not get much benefits from existing static analyzers, which suffer from mutually intermingled problems of scalability and imprecision.

In this paper, we present Loop-Sensitive Analysis (LSA) that improves the analysis scalability by enhancing the analysis precision in loops. LSA distinguishes loop iterations as many as needed by automatically choosing loop unrolling numbers during analysis. We formalize LSA in the abstract interpretation framework and prove its soundness and precision theorems using Coq. We evaluate our implementation of LSA using the analysis results of main web pages in the 5 most popular websites and those of the programs that use top 5 JavaScript libraries, and show that it outperforms the state-of-the-art JavaScript static analyzers in terms of analysis scalability. Our mechanization and implementation of LSA are both publicly available.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases JavaScript, static analysis, loops

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.735

Supplementary Material ECOOP Artifact Evaluation approved artifact available at
http://dx.doi.org/10.4230/DARTS.1.1.12

1 Introduction

The popularity of JavaScript has extended its application areas beyond simple scripts, but analyzing JavaScript applications statically is still a challenging problem. While web application developers use JavaScript to build large-scale software including games, compilers, and even operating systems, tool supports for developing them are still in a primitive stage compared to those for statically typed languages such as C and Java. Building development tools that aid programmers understand and debug programs often requires scalable and precise static analysis techniques, but extremely functional and dynamic features of JavaScript make static analysis impractical. For a function call "o`[`e`]()`", for example, because JavaScript provides first-class functions and dynamic property accesses in objects, statically estimating possible values of e often leads to an imprecise result, which in turn results in many false-positive function calls. Imprecise analysis produces false execution flows to analyze incurring much performance overhead, which makes analysis results even more imprecise.

Researchers have proposed various techniques to improve analysis precision for JavaScript web applications such as specializing specific programming patterns [23], using run-time information for determinate values [20], and combining multiple heuristic specialization methods [1]. They show that improving analysis precision significantly improves analysis

© Changhee Park and Sukyoung Ryu;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 735–756

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

performance as well. However, their static analyzers are not yet scalable enough to analyze simple applications using major libraries while they do not have any soundness proofs. Simply loading one of major libraries such as jQuery[1], Mootools[2], and Prototype[3] involves many dynamic features of JavaScript, and thus existing static analysis techniques suffer from the scalability problem due to imprecise analysis results.

In this paper, we present a novel analysis technique, *Loop-Sensitive Analysis* (LSA), which dramatically improves the analysis scalability of JavaScript applications by enhancing the analysis precision in loops. LSA distinguishes each iteration of loops with determinate loop conditions using loop contexts during analysis. It may look similar to the traditional loop unrolling but it selectively unrolls loops with determinate loop conditions and it decides the unrolling number for each loop differently and automatically during analysis. While existing loop specialization techniques [1, 23] are applicable to only special forms of loops, LSA is applicable to any forms of loops.

We formalize LSA in the abstract interpretation framework [7, 8], prove its soundness and precision theorems, and evaluate its implementation with top 5 JavaScript libraries and main web pages in the 5 most popular websites. While loop-sensitivity can be used with any form of context-sensitivity, we formally present LSA as an extension of k-CFA (Control Flow Analysis) [21] to show their relationship rigorously for language-independent programs represented by Control Flow Graphs (CFGs). We prove that LSA is sound if its base k-CFA is sound and that LSA provides more precise than or at least as precise as the analysis results of k-CFA using the proof assistant tool Coq [5]; the mechanized proofs are publicly available [12]. We implement LSA on top of an open-source JavaScript analysis framework, SAFE [13, 15]. The LSA implementation demonstrates that LSA significantly improves the analysis scalability and precision so that it can analyze all versions of jQuery, the most widely used JavaScript library, 4 of top 5 libraries, and 3 of the 5 main web pages of the most popular websites in a reasonable practical time, which outperforms the state-of-the-art JavaScript static analyzers, TAJS [1] and WALA [20], in terms of scalability.

The contributions of this paper are as follows:

- We present a novel analysis technique, LSA, which improves analysis *precision* by distinguishing each iteration of loops as many as needed during analysis. The technique is language independent and it is applicable to analysis of programs in other languages than JavaScript.
- We *formalize* LSA in the abstract interpretation framework and show how to extend k-CFA to use the technique. The formalization specifies the technical details of LSA and it is usable for formal proofs and verification.
- We provide *mechanized proofs* of the soundness and precision of LSA using the proof assistant tool, Coq [12].
- We make an LSA *implementation publicly available*; the artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). Using the implementation, we demonstrate that LSA outperforms the state-of-the-art JavaScript static analyzers in analyzing top 5 JavaScript libraries and the main web pages of the 5 most popular websites in a *scalable* way by improving analysis precision.

[1] `http://jquery.com`
[2] `http://mootools.net`
[3] `http://prototypejs.org`

```
1  jQuery.extend = function() {
2     var options, name, copy, ...
3         target = arguments[0] ...
4         i=1, length = arguments.length ...
5     if(i === length) {
6        target = this;
7        i--;
8     } ...
9     for(; i < length; i ++) { ...
10        options = arguments[i] ...
11        for (name in options) { ...
12           copy = options[name] ...
13           target[name] = copy ...
14        } ...
15     }
16  }
17  jQuery.extend({expendo: ... ,
18                   each: ... });
19  jQuery.each(...);
```

Figure 1 An excerpt from jQuery 2.1.0.

The rest of the paper is organized as follows. In Section 2, we provide a high-level idea of LSA using a motivating example. Sections 3 and 4 formally describe the concrete collecting semantics of our target programs and k-CFA, respectively. In Section 5, we present LSA as an extension of k-CFA and show its soundness and precision theorems. We evaluate a prototype implementation of LSA in terms of scalability and precision in Section 6, discuss related work in Section 7, and conclude in Section 8.

2 Motivation

In this section, after showing a running example that represents typical code patterns in JavaScript applications, we describe the scalability and precision problems in statically analyzing them. Then, we present a high-level idea of our solution, LSA.

Figure 1 shows our running example, an excerpt from the latest jQuery 2.1.0 library omitting irrelevant parts for presentation brevity. It first defines the method `jQuery.extend` (line 1) and calls it (line 17) with one object argument that has 25 properties: 4 fields and 21 methods including the method `each`. At this call site, the `extend` method extends the `jQuery` object with new properties in the argument object. Note that, in JavaScript, the value of the `arguments` object in a callee is an array object including the arguments passed by a caller [9]. In this case, because `arguments.length` is 1, the value of `target` becomes the value of `this`, the `jQuery` object, (line 6) and `i` becomes 0 (line 7) before getting into the `for` loops. Then, the subsequent loops extend the `jQuery` object by copying all the properties in the argument object one by one, and the subsequent call of `jQuery.each` (line 19) calls the function added by the `jQuery.extend` function.

While jQuery is the most widely used JavaScript library with a market share of more

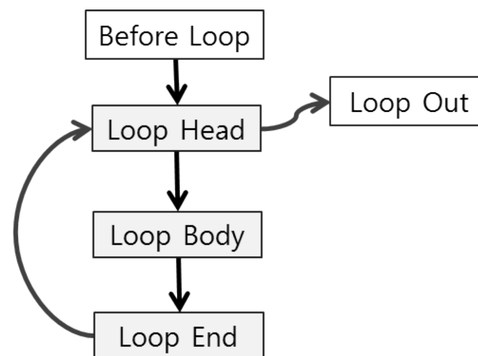


Figure 2 Control flow graph for a loop.

than 90%[4], we found that the state-of-the-art static analyzers for JavaScript applications such as SAFE [13], TAJS [18], and WALA [11] are not capable of analyzing all versions of jQuery. For example, with flow sensitivity (distinguishing different program points) [6] and call-context sensitivity (distinguishing different call sites) techniques, analysis of a simple program that just loads jQuery 2.1.0 using SAFE does not terminate in 5 hours; the analysis does not terminate either with varying sensitivities like object-sensitivity (distinguishing calls by the addresses of receiver objects) [22] and k-CFA with 1 to 10 for k (distinguishing calls by k-length call strings that represent call sequences).

We observed that this scalability problem arises due to the combination of imprecise analysis results in loops and the dynamic nature of object property names in JavaScript. Most static analyzers represent a loop as a CFG illustrated in Figure 2 and simply join the analysis results from all incoming edges into a loop-head node, which is the same as combining analysis results of all iterations. Let us revisit the code example in Figure 1. When the SAFE static analyzer analyzes the call of `jQuery.extend` on line 17, it estimates that the possible values of `options` on line 10 include the argument object passed on line 17, which approximates the possible values of `name` on line 11 as all 25 property names of the argument object. Therefore, the nested loop body on lines 12 and 13 copies the joined value of the values of 25 properties to all 25 properties in the `jQuery` object by `target[name] = copy`. Then, the analyzer estimates that the subsequent `jQuery.each` call may invoke all possible 21 functions. Because jQuery calls such methods frequently at loading time, the imprecise analysis results lead to state explosion and incur large performance overhead. We found such patterns in various applications including 9 of top 10 popular websites[5] as well as in major JavaScript libraries.

To alleviate the scalability problem, we improve analysis precision in loops by distinguishing each iteration of loops as many as needed during analysis with different loop contexts for each iteration depending on the analysis results of loop conditional expressions. It is similar to unrolling loops during analysis by finding precise unrolling counts for each loop automatically as far as the loop conditions keep definite. As for the example in Figure 1, precise unrolling counts for two loops on lines 9 and 11 should be 1 and 25, respectively. Our analysis technique indeed creates 1 and 25 different loop contexts for the loops, respectively, as long as the analysis results of loop conditional expressions are determinate for each iteration.

[4] `http://w3techs.com/technologies/overview/javascript_library/all`
[5] `http://www.alexa.com`

$$
\begin{array}{lll}
s & \in \mathbf{S} \\
n_{normal} & \in \mathbf{N_{normal}} \\
n_{call} & \in \mathbf{N_{call}} \\
n_{afterCall} & \in \mathbf{N_{afterCall}} \\
n_{entry} & \in \mathbf{N_{entry}} = \{\mathsf{n}^{\mathsf{global}}_{entry}, \dots\} \\
n_{exit} & \in \mathbf{N_{exit}} \\
n & \in \mathbf{N} = (\mathbf{N_{normal}} \uplus \mathbf{N_{call}} \uplus \mathbf{N_{afterCall}} \uplus \mathbf{N_{entry}} \uplus \mathbf{N_{exit}}) \\
e = \langle n_1, s, n_2 \rangle & \in \mathbf{E} \subseteq \mathbf{N} \times \mathbf{S} \times \mathbf{N} \\
p & \in \mathbf{P} = (\mathbf{N} \uplus \mathbf{E}) \\
callNode & \in \mathbf{N_{afterCall}} \to \mathbf{N_{call}}
\end{array}
$$

$$
\begin{array}{ll}
origin(\langle n_1, s, n_2 \rangle) & = n_1 \\
stmt(\langle n_1, s, n_2 \rangle) & = s \\
target(\langle n_1, s, n_2 \rangle) & = n_2 \\
callEdge(\langle n_1, s, n_2 \rangle) & = n_1 \in \mathbf{N_{call}} \ \wedge \ n_2 \in \mathbf{N_{entry}} \\
returnEdge(\langle n_1, s, n_2 \rangle) & = n_1 \in \mathbf{N_{exit}} \ \wedge \ n_2 \in \mathbf{N_{afterCall}} \\
normalEdge(\langle n_1, s, n_2 \rangle) & = \neg callEdge(\langle n_1, s, n_2 \rangle) \ \wedge \ \neg returnEdge(\langle n_1, s, n_2 \rangle)
\end{array}
$$

Figure 3 Program representation as a CFG.

Then, it propagates analysis results only in the loop contexts where the analysis results of the loop conditional expressions are not true to outside of loops. In this way, we can analyze loop iterations more precisely, which enables to analyze the value of `jQuery.each` as exactly the function on line 18 and consequently to analyze the following `jQuery.each` call on line 19 more precisely. In Section 6, we demonstrate that this technique can dramatically improve the analysis scalability.

3 Collecting Semantics of Programs

In this section, we formally describe the concrete collecting semantics of programs. We represent a program as a language-independent Control Flow Graph (CFG) and define its collecting semantics as a map from each *program point* (a CFG node or a CFG edge) to a set of all reachable concrete states at the point. Our formalization uses notations from Mangal *et al.*'s [17].

3.1 Program Representation

We represent a program as an interprocedural CFG $G = (\mathbf{S}, \mathbf{N}, \mathbf{E}, \mathsf{n}^{\mathsf{global}}_{entry})$ as summarized in Figure 3; $\mathbf{S}$ is a set of statements in the program; $\mathbf{N}$ is a set of nodes consisting of normal nodes $\mathbf{N_{normal}}$, call nodes $\mathbf{N_{call}}$, after-call nodes $\mathbf{N_{afterCall}}$, entry nodes $\mathbf{N_{entry}}$, and exit nodes $\mathbf{N_{exit}}$; $\mathbf{E}$ is a set of edges where an edge is a 3-tuple of a node, a statement, and another node; $\mathsf{n}^{\mathsf{global}}_{entry} \in \mathbf{N_{entry}}$ is a special entry node for the program. We use $\uplus$ to denote a disjoint union of sets.

We assume that CFGs are well-formed: a CFG has an entry node and an exit node for each function; it also has a call edge from a call node to an entry node and its corresponding return edge from an exit node to an after-call node for each function call. The function *callNode* takes an after-call node and returns its corresponding call node. The helper functions

$$\begin{array}{lrcl}
\text{(set of states)} & \tau & \in & \wp(\mathbf{State}) = \{\tau_{\mathsf{init}}, \ldots\} \\
\text{(summary map)} & \kappa_c & \in & \mathbf{A_{con}} = \mathbf{P} \rightarrow \wp(\mathbf{State}) \\
 & [\![s]\!]_c & \in & \wp(\mathbf{State}) \rightarrow \wp(\mathbf{State}) \\
 & F_{con}(\kappa_c)(n) & = & \bigcup \{\kappa_c(e) \mid n = target(e)\} \\
 & F_{con}(\kappa_c)(e) & = & [\![stmt(e)]\!]_c(\kappa_c(origin(e)))
\end{array}$$

Figure 4 Concrete domain and transfer functions.

origin, *stmt*, and *target* are accessors of an edge, and *callEdge*, *returnEdge*, and *normalEdge* are predicates that identify the kind of an edge.

3.2 Concrete Domain and Transfer Functions

Figure 4 shows the concrete domain and transfer functions of our collecting semantics. The concrete domain is a powerset of concrete states $\wp(\mathbf{State})$, where τ_{init} is a set of initial states. The collecting semantics repeatedly updates a summary map κ_c from program points $\mathbf{P}$ to sets of states using the transfer function F_{con}. For each node n, the transfer function collects the concrete states of all the incoming edges of n. For each edge e, the transfer function performs a concrete execution of the statement $stmt(e)$ using the statement transfer function $[\![\bullet]\!]_c$ on each state at the origin node of e and returns resulting states.

We assume that the set domain of concrete states is a finite complete lattice with the subset relation ordering. Then, the domain $\mathbf{A_{con}}$ of summary maps is also a complete lattice using the following order relation:

$$\forall \kappa_{c_1}, \kappa_{c_2} \in \mathbf{A_{con}} : (\kappa_{c_1} \sqsubseteq \kappa_{c_2} \iff \forall p : \kappa_{c_1}(p) \sqsubseteq \kappa_{c_2}(p)).$$

We also assume that the statement transfer function is monotone:

$$\forall s, \tau_1, \tau_2 : \tau_1 \sqsubseteq \tau_2 \implies [\![s]\!]_c(\tau_1) \sqsubseteq [\![s]\!]_c(\tau_2).$$

Then, the final collecting semantics of a program is defined as a least fixpoint of F_{con} as follows:

$$\begin{array}{l}
\kappa_{con} = \mathsf{leastFix}\ \lambda\kappa_c.(\kappa_I \sqcup F_{con}(\kappa_c)) \\
\text{where } \kappa_I(p) = \text{if } p = \mathsf{n}^{\mathsf{global}}_{entry} \text{ then } \tau_{\mathsf{init}} \text{ else } \bot.
\end{array}$$

An initial summary map κ_I maps only the global entry node to the set of initial states τ_{init} and others to $\bot$ (empty set). We can easily prove that F_{con} is monotone on summary maps :

$$\forall \kappa_{c_1}, \kappa_{c_2} : \kappa_{c_1} \sqsubseteq \kappa_{c_2} \implies F_{con}(\kappa_{c_1}) \sqsubseteq F_{con}(\kappa_{c_2})$$

and the unique least fixpoint exists by the Tarski theorem [24].

4 k-CFA Formalization

Before describing our LSA in the next section, we formalize k-CFA with the program representation of Figure 3 in this section.

4.1 Abstract Interpretation

Abstract interpretation [7, 8] is a theoretical foundation of various static analyses; it guarantees the soundness of an analysis when the analysis satisfies some required conditions. In abstract interpretation, a program analysis is a computation of monotone transfer functions on an abstract domain $\hat{D}$, which represents an approximation of a concrete domain D on which concrete programs execute. When a pair of functions α and γ satisfies the following Galois connection condition:

$$\forall x \in D, \hat{x} \in \hat{D} : \alpha(x) \sqsubseteq \hat{x} \iff x \sqsubseteq \gamma(\hat{x})$$

they provide relationships between elements in the domains: an abstraction of a concrete value x is $\alpha(x)$, and concrete values denoted by an abstract value $\hat{x}$ is $\gamma(\hat{x})$. With this condition, a concrete and monotone transfer function F and its corresponding abstract transfer function $\hat{F}$ should satisfy the following:

$$\alpha \circ F \sqsubseteq \hat{F} \circ \alpha$$

where $\circ$ denotes function composition. Once an analysis meets all the conditions above, abstract interpretation guarantees that a concrete program execution result by a least fixpoint of F and its abstract program execution result by a least fixpoint of $\hat{F}$ satisfy the following soundness property:

$$\alpha \circ \mathsf{leastFix}\ F \ \sqsubseteq\ \mathsf{leastFix}\ \hat{F}$$

which means that the analysis result soundly approximates the concrete program result.

4.2 Formal Description of k-CFA

k-CFA [21] is a call-context sensitive analysis; it distinguishes the same function body from its different call sites using the k number of call strings that represent call history. As a simple example, consider the following function calls:

```
function g() { ... }
function f() { g(); }
f();
f();
```

In 0-CFA, two calls for `f` are indistinguishable; two input states from the calls are joined at the entry of the `f` body. On the contrary, 1-CFA can distinguish the two `f` calls giving more precise analysis results for `f` than the ones from 0-CFA, but it still cannot distinguish the `g` calls at the first and the second `f` calls since it maintains only one length of call strings; likewise, 2-CFA can distinguish the `g` calls but any function calls in a deeper level. In general, k-CFA provides more precise results with the longer length of k at the expense of performance overhead due to more call contexts.

4.2.1 Analysis Domain and Transfer Functions

Figure 5 shows an analysis domain and statement transfer functions for k-CFA. An abstract domain $\hat{\mathbf{State}}$ is a finite complete lattice representing a set of abstract states where the state $\hat{\tau}_{\mathsf{init}}$ denotes an initial state when the analysis begins. The statement transfer function $[\![s]\!]$ is monotone on $\hat{\mathbf{State}}$:

$$
\begin{array}{llll}
\text{(abstract state)} & \hat{\tau} & \in & \hat{\mathbf{State}} = \{\hat{\tau}_{\mathsf{init}}, \ldots\} \\
\text{(lattice operations)} & \sqcup, \sqcap & \in & \wp(\hat{\mathbf{State}}) \to \hat{\mathbf{State}} \\
 & \bot, \top & \in & \hat{\mathbf{State}} \\
 & \sqsubseteq & \subseteq & \hat{\mathbf{State}} \times \hat{\mathbf{State}} \\
\text{(transfer functions)} & [\![s]\!] & \in & \hat{\mathbf{State}} \to \hat{\mathbf{State}} \\
(k\text{-length call string}) & \pi & \in & \mathbf{\Pi} = \{\epsilon\} \uplus \biguplus_{k \geq n \geq 1} (\mathbf{N}_{\mathbf{call}})^n \\
(k\text{-CFA annotation}) & \hat{\kappa}_c & \in & \hat{\mathbf{A}}_{\mathsf{cfa}} = (\mathbf{P} \times \mathbf{\Pi}) \to \hat{\mathbf{State}}
\end{array}
$$

(sequence operations)

$$
n \oplus \pi = n\,\pi
$$

$$
\pi \# k = \begin{cases} \epsilon & \text{if } k = 0 \\ \pi & \text{if } k > 0 \ \wedge\ |\pi| \leq k \\ n_1 \cdots n_k & \text{if } k > 0 \ \wedge\ \pi = n_1 \cdots n_k\, n_{k+1} \cdots \end{cases}
$$

Figure 5 k-CFA domain and statement transfer functions.

$$
\hat{F}_{\mathsf{cfa}}(\hat{\kappa}_c)(\langle n, \pi \rangle) = \bigsqcup \{\hat{\kappa}_c(\langle e, \pi \rangle) \mid n = \mathit{target}(e)\}
$$

$$
\hat{F}_{\mathsf{cfa}}(\hat{\kappa}_c)(\langle e, \pi \rangle) = \begin{cases} [\![\mathit{stmt}(e)]\!](\hat{\kappa}_c(\langle \mathit{origin}(e), \pi \rangle)) & \text{if } \mathit{normalEdge}(e) \\ \bigsqcup \{[\![\mathit{stmt}(e)]\!](\hat{\kappa}_c(\langle \mathit{origin}(e), \pi_1 \rangle)) \mid & \text{if } \mathit{callEdge}(e) \\ \qquad \pi = (\mathit{origin}(e) \oplus \pi_1)\#k\} & \\ [\![\mathit{stmt}(e)]\!](\hat{\kappa}_c(\langle \mathit{origin}(e), & \text{if } \mathit{returnEdge}(e) \\ \qquad (\mathit{callNode}(\mathit{target}(e)) \oplus \pi)\#k \rangle)) & \end{cases}
$$

Figure 6 Transfer functions on k-CFA annotations.

$$
\forall s, \tau_1, \tau_2 : \tau_1 \sqsubseteq \tau_2 \Longrightarrow [\![s]\!](\tau_1) \sqsubseteq [\![s]\!](\tau_2)
$$

and it abstractly executes the statement s with an input state and produces an output state. A k-length call string π is a sequence of call nodes with the maximum length k and ϵ denotes the empty sequence. We write $n \oplus \pi$ to denote prepending a new call node n to a sequence π and $\pi \# k$ to denote the k-length prefix of π. Finally, the k-CFA annotation $\hat{\kappa_c}$ maps a pair of a node or an edge and a call string to its corresponding abstract state: $\hat{\kappa_c}(\langle p, \pi \rangle)$ gives an approximate input state at the program point p in the call context represented by the call string π. The domain $\hat{\mathbf{A}}_{\mathsf{cfa}}$ of k-CFA annotations is a complete lattice using the following order relation:

$$
\forall \hat{\kappa}_{c_1}, \hat{\kappa}_{c_2} : (\hat{\kappa}_{c_1} \sqsubseteq \hat{\kappa}_{c_2} \Longleftrightarrow \forall p, \pi : \hat{\kappa}_{c_1}(\langle p, \pi \rangle) \sqsubseteq \hat{\kappa}_{c_2}(\langle p, \pi \rangle)).
$$

4.2.2 Control Flow Analysis

Figure 6 shows the definition of the transfer function $\hat{F}_{\mathsf{cfa}}$ on k-CFA annotations. The transfer function takes a k-CFA annotation and returns an updated k-CFA annotation by transferring the analyzed state of a preceding node or edge to the current node or edge; for call and return edges, it updates call strings appropriately. Note that while a join operator in the transfer function for call edges is necessary to join all analysis results from calls with the

same call strings but different call history at the entry of the target function, the operator is not needed in the transfer function for return edges because analysis results from different call history are already combined at the entry of the function by the corresponding call edges. Then, k-CFA computes the fixpoint of $\hat{F}_{\mathsf{cfa}}$, $\hat{\kappa}_{\mathsf{cfa}}$, which contains final analysis results for all nodes and edges in a CFG with the maximum k-length call contexts:

$$\hat{\kappa}_{\mathsf{cfa}} = \mathsf{leastFix}\ \lambda\hat{\kappa}_c.(\hat{\kappa}_I \sqcup \hat{F}_{\mathsf{cfa}}(\hat{\kappa}_c))$$
$$\text{where } \hat{\kappa}_I(\langle p, \pi\rangle) = \text{if } \langle p, \pi\rangle = \langle \mathsf{n}^{\mathsf{global}}_{entry}, \epsilon\rangle \text{ then } \hat{\tau}_{\mathsf{init}} \text{ else } \bot.$$

The analysis begins with the initial state $\hat{\tau}_{\mathsf{init}}$ at the global entry node in the empty call context, and it propagates the abstract state from the initial node to all reachable nodes and edges via $\hat{F}_{\mathsf{cfa}}$. It is obvious from the order relation of $\hat{A}_{\mathsf{cfa}}$ that $\hat{F}_{\mathsf{cfa}}$ is monotone on k-CFA annotations:

$$\forall \hat{\kappa}_{c_1}, \hat{\kappa}_{c_2} : \hat{\kappa}_{c_1} \sqsubseteq \hat{\kappa}_{c_2} \Longrightarrow \hat{F}_{\mathsf{cfa}}(\hat{\kappa}_{c_1}) \sqsubseteq \hat{F}_{\mathsf{cfa}}(\hat{\kappa}_{c_2}).$$

Then, a unique least fixpoint of $\hat{F}_{\mathsf{cfa}}$ in a complete lattice domain exists [24] and the computation terminates since the domains of $\hat{F}_{\mathsf{cfa}}$ are all finite.

4.3 Soundness

We assume that k-CFA is a sound approximation of the collecting semantics presented in Section 3:

- Galois connection: Functions $\alpha_s \in \wp(\mathbf{State}) \to \hat{\mathbf{State}}$ and $\gamma_s \in \hat{\mathbf{State}} \to \wp(\mathbf{State})$ exist.
- $\alpha_s(\tau_{\mathsf{init}}) \sqsubseteq \hat{\tau}_{\mathsf{init}}$: $\hat{\tau}_{\mathsf{init}}$ is a sound approximation of τ_{init}.
- $\forall s \in \mathbf{S} : \alpha_s \circ [\![s]\!]_c \sqsubseteq [\![s]\!] \circ \alpha_s$: The abstract function $[\![s]\!]$ is a sound approximation of $[\![s]\!]_c$ for a statement s.
- Galois connection: Functions $\alpha_c \in \mathbf{A_{con}} \to \hat{\mathbf{A}}_{\mathbf{cfa}}$ and $\gamma_c \in \hat{\mathbf{A}}_{\mathbf{cfa}} \to \mathbf{A_{con}}$ exist.
- $\alpha_c \circ F_{con} \sqsubseteq \hat{F}_{\mathsf{cfa}} \circ \alpha_c$: The abstract function $\hat{F}_{\mathsf{cfa}}$ is a sound approximation of F_{con}.

Then, by the monotonicity of F_{con} and $\hat{F}_{\mathsf{cfa}}$, abstract interpretation guarantees the soundness of k-CFA:

$$\alpha_c(\kappa_{con}) \ \sqsubseteq\ \hat{\kappa}_{\mathsf{cfa}}.$$

We use the definitions of κ_{con} and $\hat{\kappa}_{\mathsf{cfa}}$, least fixpoints of F_{con} and $\hat{F}_{\mathsf{cfa}}$, respectively defined in previous sections for the same well-formed program p. In the next section, we use the above assumptions to show that our new analysis technique is also sound when we apply it to k-CFA.

5 Loop-Sensitive Analysis

Now, we extend k-CFA with loop-sensitivity. In k-CFA, a loop head node is simply a normal node with two incoming and two outgoing edges as shown in Figure 2. According to the node transfer function $\hat{F}_{\mathsf{cfa}}$, the abstract state of a loop head node is a join of abstract states from the two incoming edges, which effectively combines analysis results of all iterations incurring large precision losses. The main idea of LSA is to distinguish each loop iteration using *loop strings* like call strings in k-CFA to reduce precision losses. The novelty of our work lies in applying the sensitivity-based analysis technique to loops using loop strings and formally proving the soundness of the analysis in the abstract interpretation framework. Unlike the traditional loop unrolling, LSA automatically determines the unrolling number of each loop during analysis using the analysis results of loop conditional expressions, which immensely enhances the analysis performance.

$$
\begin{array}{lll}
n_{lhead} & \in & \mathbf{N_{lhead}} \\
n_{lend} & \in & \mathbf{N_{lend}} \\
n_{out} & \in & \mathbf{N_{lout}} \\
n_{break} & \in & \mathbf{N_{break}} \\
n_{continue} & \in & \mathbf{N_{continue}} \\
n_{return} & \in & \mathbf{N_{return}} \\
n & \in & \mathbf{N} = (\mathbf{N_{normal}} \uplus \mathbf{N_{call}} \uplus \mathbf{N_{afterCall}} \uplus \mathbf{N_{entry}} \uplus \mathbf{N_{exit}} \uplus \\
 & & \quad \mathbf{N_{lhead}} \uplus \mathbf{N_{lend}} \uplus \mathbf{N_{lout}} \uplus \mathbf{N_{break}} \uplus \mathbf{N_{continue}} \uplus \mathbf{N_{return}})
\end{array}
$$

$$
\begin{array}{l}
\mathit{loopHead} \in \mathbf{N} \to \mathbf{N_{lhead}} \\
\mathit{loopHeads} \in \mathbf{N} \to \wp(\mathbf{N_{lhead}}) \\
\mathit{loopInEdge}(\langle n_1, s, n_2 \rangle) = n_1 \in \mathbf{N_{normal}} \wedge n_2 \in \mathbf{N_{lhead}} \\
\mathit{loopIterEdge}(\langle n_1, s, n_2 \rangle) = n_1 \in \mathbf{N_{lend}} \wedge n_2 \in \mathbf{N_{lhead}} \\
\mathit{loopIterEdge}(\langle n_1, s, n_2 \rangle) = n_1 \in \mathbf{N_{continue}} \wedge n_2 \in \mathbf{N_{lhead}} \\
\mathit{loopOutEdge}(\langle n_1, s, n_2 \rangle) = n_1 \in \mathbf{N_{lhead}} \wedge n_2 \in \mathbf{N_{lout}} \\
\mathit{loopBreakEdge}(\langle n_1, s, n_2 \rangle) = n_1 \in \mathbf{N_{break}} \wedge n_2 \in \mathbf{N_{lout}} \\
\mathit{loopReturnEdge}(\langle n_1, s, n_2 \rangle) = n_1 \in \mathbf{N_{return}} \wedge n_2 \in \mathbf{N_{exit}} \\
\mathit{normalEdge}(\langle n_1, s, n_2 \rangle) = \neg \mathit{callEdge}(\langle n_1, s, n_2 \rangle) \wedge \neg \mathit{returnEdge}(\langle n_1, s, n_2 \rangle) \wedge \\
\quad \neg \mathit{loopInEdge}(\langle n_1, s, n_2 \rangle) \wedge \neg \mathit{loopIterEdge}(\langle n_1, s, n_2 \rangle) \wedge \\
\quad \neg \mathit{loopOutEdge}(\langle n_1, s, n_2 \rangle) \wedge \neg \mathit{loopBreakEdge}(\langle n_1, s, n_2 \rangle) \wedge \\
\quad \neg \mathit{loopReturnEdge}(\langle n_1, s, n_2 \rangle)
\end{array}
$$

Figure 7 Program representation with loop-sensitivity.

5.1 Formal Description

We formally describe how to extend sound k-CFA to LSA while preserving the soundness of the analysis.

5.1.1 Program Representation

Without loss of generality, we assume that we can rewrite all loop constructs in a target program as "`while(`e`)`s" where the evaluation of e does not have any side effects and CFGs for all loops conform to the structure in Figure 2. Note that for-in loops in JavaScript cannot be rewritten into while loops in general since the iteration order is implementation specific according to the ECMAScript standard [9]. However, real-world implementations in major browsers such as Internet Explorer, Safari, Firefox, and Chrome use the same iteration order; for instance, in `for(x in obj)`, they use the same order that the properties were added to the object `obj`. Thus, we use this order for the JavaScript cases. Figure 7 presents the extension of CFGs in Figure 3 for loop-sensitivity. In addition to call, after-call, entry, and exit nodes introduced in Figure 3, we refine nodes further to distinguish nodes in loops as loop-head, loop-end, loop-out, break, continue, and return nodes. The functions *loopHead* and *loopHeads* take a node in a loop and return its innermost loop-head node and a set of all its enclosing loop-head nodes, respectively. We also refine edges further to distinguish loop-related edges as follows:

- loop-in edge: an edge from a normal node outside a loop to a loop-head node
- loop-iter edge: an edge from a loop-end node or a continue node to a loop-head node
- loop-out edge: an edge from a loop-head node to a loop-out node
- loop-break edge: an edge from a break node inside a loop to a loop-out node
- loop-return edge: an edge from a return node inside a loop to an exit node

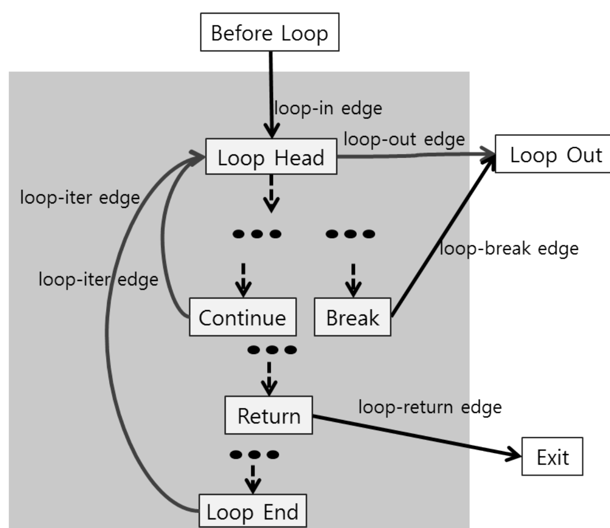


Figure 8 Loop-related nodes and edges for LSA.

(i-depth j-length loop string)

$$\psi \in \mathbf{\Psi} = \{\epsilon\} \uplus \biguplus_{i \geq n \geq 1} (\mathbf{\Pi} \times \mathbf{N_{lhead}} \times \mathbb{N}_{\{0..j\}})^n$$

(abstract value for loop conditional expression)

$$check(s, \hat{\tau}) \in \mathbf{\hat{Bool}} = \{\bot_b, \hat{\mathsf{true}}, \hat{\mathsf{false}}, \top_b\}$$

(context)

$$\phi = \langle \pi, \psi \rangle \in \mathbf{\Phi} = \mathbf{\Pi} \times \mathbf{\Psi}$$

(LSA annotation)

$$\hat{\kappa}_l \in \hat{\mathbf{A}}_{\mathsf{lsa}} = (\mathbf{P} \times \mathbf{\Phi}) \rightarrow \mathbf{\hat{State}}$$

Figure 9 LSA domain and transfer functions.

The functions *loopInEdge*, *loopIterEdge*, *loopOutEdge*, *loopBreakEdge*, and *loopReturnEdge* are predicates identifying the kind of an edge. Note that this extension does not change CFG structures but just refine the kinds of nodes and edges; all loop-related nodes and edges are normal nodes and edges in k-CFA. Figure 8 illustrates loop-related nodes and edges: a shaded box denotes a loop, dashed edges denote normal edges, and solid edges denote loop-related edges.

5.1.2 Analysis Domain and Transfer Functions

Figure 9 shows the extension of the analysis domain and transfer functions in Figure 5 for loop-sensitivity. A loop string ψ is a maximum i-length sequence of *loop contexts*; for a loop context $\langle \pi, n_{lhead}, m \rangle$, π is a call context where this loop context is introduced, n_{lhead} is a loop-head node that introduces this loop context, and m is a loop iteration count where $m \in \mathbb{N}_{\{0..j\}}$; $\mathbb{N}_{\{0..j\}}$ is the set of natural numbers between 0 and j. When loop iterations are indistinguishable because the value of the loop conditional expression may be both true and false, we call such contexts *join loop contexts* and denote them by $m = 0$. We call LSA using such "i-depth j-length loop strings" and k-length call strings $\langle i, j, k \rangle$-LSA. The values

of i and j are predefined as k in k-CFA. Intuitively, i and j mean the maximum depth of distinguishable nested loops and the maximum number of distinguishable iterations in a loop, respectively. For instance, if $i = 0$, LSA is the same as k-CFA and $\langle 2, 10, k\rangle$-LSA can distinguish up to 10 iterations in each loop with 2-level nested loops. As a design choice, we require that $j \geq 1$ while $i \geq 0$ and $k \geq 0$. Note that the value of j is effective only when $i \geq 1$.

The function *check* checks if a boolean expression s evaluates to an error ($\bot_b$), true, false, or both ($\top_b$) in a given abstract state $\hat{\tau}$; we use this function to update loop contexts depending on the values of loop conditionals as we explain later in this section. A context ϕ is a pair of a call string and a loop string, and an LSA annotation $\hat{\kappa}_l$ maps a pair of a node or an edge and a context to its corresponding abstract state: $\hat{\kappa}_l(\langle p, \phi\rangle)$ gives an approximate input state at the program point p in the context ϕ. Note that a call string in a context ϕ denotes different information from a call string in a loop context element $\langle \pi, n_{lhead}, m\rangle$; the former is for k-CFA on which LSA is based, and the latter is to serve as a call context where the loop is introduced, which is necessary to change the current loop context properly when the flow changes by return statements in loops as we present later in this section. The domain $\hat{\mathbf{A}}_{\mathsf{lsa}}$ of LSA annotations is a complete lattice using the following order relation:

$$\forall \hat{\kappa}_{l_1}, \hat{\kappa}_{l_2} : (\hat{\kappa}_{l_1} \sqsubseteq \hat{\kappa}_{l_2} \iff \forall p, \phi : \hat{\kappa}_{l_1}(\langle p, \phi\rangle) \sqsubseteq \hat{\kappa}_{l_2}(\langle p, \phi\rangle)).$$

5.1.3 Loop-Sensitive Analysis

As with k-CFA, $\langle i, j, k\rangle$-LSA computes the least fixpoint of the transfer function $\hat{F}_{\mathsf{lsa}}$ on LSA annotations as follows:

$$\hat{\kappa}_{\mathsf{lsa}} = \mathsf{leastFix}\ \lambda\hat{\kappa}_l.(\hat{\kappa}_I \sqcup \hat{F}_{\mathsf{lsa}}(\hat{\kappa}_l))$$
$$\text{where } \hat{\kappa}_I(\langle p, \langle \pi, \psi\rangle\rangle) = \text{if } \langle p, \langle \pi, \psi\rangle\rangle = \langle \mathsf{n}^{\mathsf{global}}_{entry}, \langle \epsilon, \epsilon\rangle\rangle \text{ then } \hat{\tau}_{\mathsf{init}} \text{ else } \bot$$

where the analysis starts with the initial state $\hat{\tau}_{\mathsf{init}}$ at the global entry node in the empty call and loop contexts. Like $\hat{F}_{\mathsf{cfa}}$ in Figure 6, the definition of $\hat{F}_{\mathsf{lsa}}$ consists of node and edge transfer functions. The transfer functions for nodes and normal, call, and return edges remain the same as in k-CFA except that a context is now a pair of a call string and a loop string instead of a single call string. Due to the space limitation, we present core parts of the transfer functions for loop-related edges, and refer the interested readers to a companion report [12] for the full definition.

Loop-in edge. Figure 10 shows the definition of $\hat{F}_{\mathsf{lsa}}$ for loop-in edges. For presentation brevity, we omit universal quantifiers binding meta variables in obvious cases. For example, $\psi = \langle \pi_1, n_{lhead}, m\rangle \oplus \psi_1$ in the second rule is equal to $\forall \pi_1, n_{lhead}, m, \psi_1 : \psi = \langle \pi_1, n_{lhead}, m\rangle \oplus \psi_1$.

The first rule states that when $i = 0$, LSA is the same as k-CFA; the abstract states from the origin node of the edge propagate through the edge without changing the current loop string: $[\![stmt(e)]\!](\hat{\kappa}_l(\langle origin(e), \langle \pi, \psi\rangle\rangle))$. The function $\hat{F}_{\mathsf{lsa}}$ has similar rules for all the other loop-related edges, and we omit them in this paper for brevity.

The second rule specifies the case when the depth of the current loop string $|\psi|$ reaches the maximum depth i not by the current loop but by an outer loop: it is either from a different call context $\pi \neq \pi_1$ or from a different loop $n_{lhead} \neq target(e)$. Then, LSA simply propagates the abstract states from the origin node of the edge. Other loop-related edges also have similar rules.

The third and fourth rules describe the case with a join loop context, where the iteration count of the innermost loop string is 0. The third rule is when an outer loop creates the

$$\hat{F}_{\mathsf{lsa}}(\hat{\kappa}_l)(\langle e, \langle \pi, \psi\rangle\rangle) \quad \text{if } \mathit{loopInEdge}(e) \; = \begin{cases} [\![\mathit{stmt}(e)]\!](\hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi\rangle\rangle)) & \text{if } i = 0 \\ [\![\mathit{stmt}(e)]\!](\hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi\rangle\rangle)) & \text{if } |\psi| = i \;\wedge\; \psi = \langle \pi_1, n_{lhead}, m\rangle \oplus \psi_1 \;\wedge \\ & \quad (\pi \neq \pi_1 \vee\; n_{lhead} \neq \mathit{target}(e)) \\ [\![\mathit{stmt}(e)]\!](\hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi\rangle\rangle)) & \text{if } |\psi| < i \;\wedge\; \psi = \langle \pi_1, n_{lhead}, 0\rangle \oplus \psi_1 \;\wedge \\ & \quad (\pi \neq \pi_1 \vee n_{lhead} \neq \mathit{target}(e)) \\ [\![\mathit{stmt}(e)]\!](\hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi_1\rangle\rangle)) & \text{if } \psi = \langle \pi, \mathit{target}(e), 0\rangle \oplus \psi_1 \;\wedge \\ & \quad \forall \langle \pi_1, n_{lhead}, m\rangle, \psi_2 : \\ & \qquad \langle \pi_1, n_{lhead}, m\rangle \oplus \psi_2 = \psi_1 \wedge m \neq 0 \\ [\![\mathit{stmt}(e)]\!](\hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi_1\rangle\rangle)) & \text{if } \psi = \langle \pi, \mathit{target}(e), 1\rangle \oplus \psi_1 \end{cases}$$

Figure 10 Transfer functions for loop-in edges.

$$\hat{F}_{\mathsf{lsa}}(\hat{\kappa}_l)(\langle e, \langle \pi, \psi\rangle\rangle) \quad \text{if } \; \mathit{loopIterEdge}(e) \;\wedge \psi = \langle \pi, \mathit{target}(e), m\rangle \oplus \psi_1 \; = \begin{cases} [\![\mathit{stmt}(e)]\!](\hat{\tau}_1) \sqcup & \text{if } \; m = j \;\wedge \\ [\![\mathit{stmt}(e)]\!](\hat{\tau}_2) & \quad \hat{\tau}_1 = \hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \langle \pi, \mathit{target}(e), m{-}1\rangle \oplus \psi_1\rangle\rangle) \;\wedge \\ & \quad \hat{\tau}_2 = \hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi\rangle\rangle) \\ [\![\mathit{stmt}(e)]\!](\hat{\tau}_1) \sqcup & \text{if } \; 2 \leq m \leq j{-}1 \;\wedge \\ [\![\mathit{stmt}(e)]\!](\hat{\tau}_2) & \quad \hat{\tau}_1 = \hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \langle \pi, \mathit{target}(e), m{-}1\rangle \oplus \psi_1\rangle\rangle) \;\wedge \\ & \quad \hat{\tau}_2 = \hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi\rangle\rangle) \;\wedge \\ & \quad (\mathit{check}(\mathit{stmt}(e), \hat{\tau}_1) = \top_b \;\vee\; \mathit{check}(\mathit{stmt}(e), \hat{\tau}_2) = \top_b) \\ [\![\mathit{stmt}(e)]\!](\hat{\tau}) & \text{if } \; 2 \leq m \leq j{-}1 \;\wedge \\ & \quad \hat{\tau} = \hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \langle \pi, \mathit{target}(e), m{-}1\rangle \oplus \psi_1\rangle\rangle) \;\wedge \\ & \quad (\mathit{check}(\mathit{stmt}(e), \hat{\tau}) = \widehat{\mathsf{true}} \;\vee\; \mathit{check}(\mathit{stmt}(e), \hat{\tau}) = \widehat{\mathsf{false}}) \end{cases}$$

Figure 11 Transfer functions for loop-iter edges.

join loop context and the fourth rule is when the current loop creates it. The condition $\forall \langle \pi_1, n_{lhead}, m\rangle, \psi_2 : \langle \pi_1, n_{lhead}, m\rangle \oplus \psi_2 = \psi_1 \wedge m \neq 0$ in the fourth rule prevents prepending another join loop context when the outer loop of the current one already creates a join loop context. In a join loop context, LSA joins the analysis results from loop-in and loop-iter edges as the final analysis result of the loop. Note that join loop contexts have the same effect as preventing loops with non-deterministic loop conditions from unrolling and thus avoiding unnecessary fixpoint computation.

The fifth rule addresses the case for analyzing loop bodies. It prepends a new loop context $\langle \pi, \mathit{target}(e), 1\rangle$ denoting the first iteration of the loop in the same call context to the current loop string, and propagates the abstract states from the origin node of the edge in the context before the loop through the edge: $[\![\mathit{stmt}(e)]\!](\hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi_1\rangle\rangle))$.

Loop-iter edge. Figure 11 shows a partial definition of $\hat{F}_{\mathsf{lsa}}$ for loop-iter edges, which propagate abstract states resulting from loop iterations to a loop-head node $\mathit{target}(e)$ introducing new loop contexts whenever necessary. We omit the rules similar to the ones for loop-in edges for brevity.

$$\begin{aligned}
&\hat{F}_{\mathsf{lsa}}(\hat{\kappa}_l)(\langle e, \langle \pi, \psi \rangle\rangle) \quad \text{if } loopOutEdge(e) \ \wedge\ i \neq 0 \ \ \wedge\ |\psi| < i \\
&\quad = \bigsqcup \{ [\![stmt(e)]\!](\hat{\tau}) \mid 1 \leq m \leq j \ \wedge\ \hat{\tau} = \hat{\kappa}_l(\langle origin(e), \langle \pi, \langle \pi, origin(e), m\rangle \oplus \psi \rangle\rangle) \ \wedge \\
&\qquad\qquad \hat{\mathsf{false}} \sqsubseteq check(stmt(e), \hat{\tau}) \}
\end{aligned}$$

$$\begin{aligned}
&\hat{F}_{\mathsf{lsa}}(\hat{\kappa}_l)(\langle e, \langle \pi, \psi \rangle\rangle) \quad \text{if } loopBreakEdge(e) \ \wedge\ i \neq 0 \ \wedge\ |\psi| < i \\
&\quad = \bigsqcup \{ [\![stmt(e)]\!](\hat{\tau}) \mid 1 \leq m \leq j \ \wedge \\
&\qquad\qquad \hat{\tau} = \hat{\kappa}_l(\langle origin(e), \langle \pi, \langle \pi, loopHead(origin(e)), m\rangle \oplus \psi \rangle\rangle) \}
\end{aligned}$$

$$\begin{aligned}
&\hat{F}_{\mathsf{lsa}}(\hat{\kappa}_l)(\langle e, \langle \pi, \psi \rangle\rangle) \quad \text{if } loopReturnEdge(e) \ \wedge\ i \neq 0 \ \wedge\ |\psi| < i \\
&\quad = \bigsqcup \{ [\![stmt(e)]\!](\hat{\tau}) \mid \psi_1 \in \mathbf{\Psi} \ \wedge\ n_{lhead_2} \in loopHeads(origin(e)) \ \wedge \\
&\qquad\qquad \langle \pi, n_{lhead_2}, m_2 \rangle \in \psi_1 \ \wedge \\
&\qquad\qquad 1 \leq m_2 \leq j \ \wedge\ \hat{\tau} = \hat{\kappa}_l(\langle origin(e), \langle \pi, \psi_1 || \psi \rangle\rangle) \}
\end{aligned}$$

Figure 12 Transfer functions for loop-out, loop-break, and loop-return edges.

The first rule specifies the case when the iteration count m reaches the maximum length j. In this case, all further analysis results from the current iteration in the same loop context are joined with the analysis result from the previous iteration. Note that i and j in $\langle i, j, k\rangle$-LSA with the definition of $\hat{F}_{\mathsf{lsa}}$ ensure that the analysis terminates using a finite loop-string domain.

The second and third rules are for the cases when the iteration count m does not reach the maximum length j. The second rule is when the analysis result of a loop conditional expression $stmt(e)$ in the current iteration is $\top_b$; because all further iterations from the current one are indistinguishable, the analysis results are combined with the analysis result from the previous iteration. In the third rule, because the analysis result of a loop conditional expression in the previous iteration is either true or false, it does not need to join analysis results.

Loop-out edge. Figure 12 presents representative rules for the remaining loop-related edges. The first rule specifies that a loop-out edge propagates an abstract state from a loop to a loop-out node removing the current loop context from the loop string only when the analysis result of the loop conditional expression is false or $\top_b$: $\hat{\mathsf{false}} \sqsubseteq check(stmt(e), \hat{\tau})$. By this condition, when the analysis result of a loop conditional expression is true, a loop-out edge does not propagate the abstract state from a loop to a loop-out node, which improves the analysis precision of nodes after loops.

Loop-break edge. As the second rule in Figure 12 specifies, a loop-break edge propagates an abstract state from a break node inside a loop to a loop-out node. Since the evaluation of the `break` statement breaks out of a loop, the rule reverts the loop string to the one before the current loop by finding the innermost loop-head node with *loopHead* and reverting to the loop context just before it.

Loop-return edge. The third rule in Figure 12 states that a loop-return edge propagates an abstract state from a return node inside a loop to an exit node of a function enclosing the loop. Because the evaluation of the `return` statement breaks out all the enclosing loops and returns from the enclosing function, the rule reverts the loop string to the one before the current function is called by finding all the enclosing loop-head nodes with *loopHeads*

and reverting to the loop context just before the call of the current function. Using the concatenation operator $||$, the rule joins all analysis results of the return node in contexts $\langle \pi, \psi_1 || \psi \rangle$ where ψ_1 contains loop contexts with the current call string π and the enclosing loop-head nodes of the return node: $n_{lhead_2} \in loopHeads(origin(e))$.

We prove that $\hat{F}_{\mathsf{lsa}}$ is monotone on the domain of $\hat{\kappa}_l$.

▶ **Theorem 1.** *(Monotonicity of $\hat{F}_{\mathsf{lsa}}$)*
$\forall \hat{\kappa}_{l_1}, \hat{\kappa}_{l_2} : \hat{\kappa}_{l_1} \sqsubseteq \hat{\kappa}_{l_2} \Longrightarrow \hat{F}_{\mathsf{lsa}}(\hat{\kappa}_{l_1}) \sqsubseteq \hat{F}_{\mathsf{lsa}}(\hat{\kappa}_{l_2})$.

Then, the unique least fixpoint of $\hat{F}_{\mathsf{lsa}}$ exists, and the computation terminates since the domains of $\hat{F}_{\mathsf{lsa}}$ are finite.

5.2 Soundness and Precision

This section shows that $\langle i, j, k \rangle$-LSA extending sound k-CFA is also sound and it is more precise than or as precise as k-CFA. We proved all the theorems in this paper using the proof assistant tool Coq [5] and the Coq library of lattices[6]; the mechanized proofs are publicly available [12].

5.2.1 Soundness

We define translation functions between concrete summary maps and LSA annotations $\alpha : \mathbf{A_{con}} \to \hat{\mathbf{A}}_{\mathbf{lsa}}$ and $\gamma : \hat{\mathbf{A}}_{\mathbf{lsa}} \to \mathbf{A_{con}}$ as follows:

$$\alpha(\kappa_c) = \lambda(\langle p, \phi \rangle).\alpha_s(\kappa_c(p) \sqcap statesInCtxt(\phi))$$
$$\gamma(\hat{\kappa}_l) = \bigsqcup \{\kappa_c \mid \alpha(\kappa_c) \sqsubseteq \hat{\kappa}_l\}$$

where the helper function $statesInCtxt : \mathbf{\Phi} \to \wp(\mathbf{State})$ provides a set of all possible concrete states in a given context. For a given context ϕ, the meet of all reachable states $\kappa_c(p)$ without considering contexts and possibly unreachable states $statesInCtxt(\phi)$ in the context ϕ denotes a set of all reachable states in the context. Then, α and γ satisfy the Galois connection condition:

▶ **Theorem 2.** *(Galois connection)*
$\forall \kappa_c \in \mathbf{A_{con}}, \hat{\kappa}_l \in \hat{\mathbf{A}}_{\mathbf{lsa}} : \alpha(\kappa_c) \sqsubseteq \hat{\kappa}_l \Longleftrightarrow \kappa_c \sqsubseteq \gamma(\hat{\kappa}_l)$.

We assume that all reachable states in a context ϕ are subsumed by the join of execution results from all reachable states in the preceding contexts of ϕ. Then, we can prove the soundness of the transfer function:

▶ **Theorem 3.** *(Soundness of transfer functions)*
$\alpha \circ F_{con} \sqsubseteq \hat{F}_{\mathsf{lsa}} \circ \alpha$.

Finally, with all the theorems above, the abstract interpretation framework guarantees the soundness of LSA:

$$\alpha(\kappa_{con}) \sqsubseteq \hat{\kappa}_{\mathsf{lsa}}.$$

[6] http://raweb.inria.fr/2006/Raweb/lande/uid20.html

Table 1 Analysis results of SAFE, SAFE_{lsa}, TAJS, and WALA within the timeout of 5 hours.

Group (Number of programs)	Success			
	SAFE	SAFE_{lsa}	TAJS	WALA
jQuery 1.0.0~2.1.0 (14)	0	14	11	3
Modernizr 2.8.3 (1)	1	1	1	1
BootStrap 3.3.0 (1)	0	0	0	0
Mootools 1.5.1 (1)	0	1	0	0
Prototype 1.7.2 (1)	0	1	0	0
BENCH (61)	0	37	20	0
SLICE (61)	0	53	33	0
WEBSITE (5)	0	3	0	0

5.2.2 Precision

To compare the analysis results of k-CFA and $\langle i, j, k\rangle$-LSA, we define a translation function from LSA annotations to CFA annotations $\eta : \hat{\mathbf{A}}_{\mathbf{lsa}} \to \hat{\mathbf{A}}_{\mathbf{cfa}}$ as follows:

$$\eta(\hat{\kappa}_l) = \lambda(\langle p, \pi\rangle). \bigsqcup \{\hat{\kappa}_l(\langle p, \langle \pi, \psi\rangle\rangle) \mid \psi \in \Psi\}.$$

It simply translates LSA contexts with the same call string (possibly with different loop strings) to a CFA context by joining the LSA contexts. Then, the analysis results from LSA are more precise than or at least as precise as the ones from CFA at all program points:

▶ **Theorem 4.** *(Precision)* $\eta(\hat{\kappa}_{\text{lsa}}) \sqsubseteq \hat{\kappa}_{\text{cfa}}$

6 Evaluation

In this section, we evaluate our technique in two respects, scalability and precision, using SAFE_{lsa}, an extension of an open-source framework SAFE [13, 15] that statically analyzes JavaScript web applications with modeling of various browser environments. We performed all experiments on a Mac OS X x64 machine with 3.4GHz Intel Core i7 CPU and 16GB Memory, and we used 30-depth 1000-length loop strings and 10-CFA for LSA. Even though we use big numbers for i and j for $\langle i, j, k\rangle$-LSA to ensure termination of analyses, we found that actual analyses create much smaller numbers of loop contexts without incurring much overhead. We also used 10-CFA for the experiment with SAFE for comparison.

6.1 Scalability

We evaluate the scalability of LSA by comparing the analysis results of SAFE_{lsa} with those of state-of-the-art static analyzers, SAFE, TAJS, and WALA; we refer the interested readers to Section 7 for more detailed explanation of the analyzers. For fair comparison, we used the latest versions of SAFE and WALA from their open-source repositories except for jQuery as we explain below and the specialized version for TAJS [18]. Similar to the experiments of TAJS [1], we measured how many target subjects each analyzer successfully analyzes within the timeout of 5 hours with normal analysis results. Table 1 summarizes the experimental results.

The first column in Table 1 shows target subject groups we experimented with and the numbers of programs in each group. The first 5 groups are simple programs that just load

one of the top 5 JavaScript libraries according to W3Techs[7]. Because jQuery has a more than 90% market share and many web applications still use old versions of jQuery even with newer versions, we analyze all 14 released versions of jQuery (1.0.0 $\sim$ 1.11.0, 2.0.0 $\sim$ 2.1.0) while we analyze only the latest versions for other libraries. The BENCH and SLICE groups are benchmarks from the experiments of TAJS [1]; the authors collected 71 programs from a jQuery tutorial[8] that perform simple operations using jQuery 1.10.0, and they compared analysis results of using the entire jQuery (BENCH) and using its sliced versions (SLICE). Note that our experiments have excluded 10 programs from the original 71 benchmarks because some soundness bugs in TAJS affected analysis results on the 10 programs. One such bug is that the loop specialization mechanism in TAJS misses analysis flows in the presence of the `return` statement in loops, which formal verification like our Coq mechanization can surely detect. The last group WEBSITE contains main web pages of the 5 most popular websites, `google.com`, `facebook.com`, `youtube.com`, `baidu.com`, and `yahoo.com`, according to the Alexa website[9].

The second to last columns in Table 1 show the successful analysis results. While SAFE analyzes only one program that loads Modernizr[10], $\text{SAFE}_{\mathsf{lsa}}$ performs the best: it analyzes all programs that load 14 versions of jQuery and the latest versions of Modernizr, Mootools, and Prototype; it analyzes 37, 53, and 3 programs in the BENCH, SLICE, and WEBSITE groups, respectively. Note that because TAJS does not support ES5 getters and setters, it cannot analyze jQuery version 2.x. Also, while WALA can analyze 3 versions of jQuery using the dynamic determinacy technique [20], the technique is not available from its latest open-source repository. Thus, Table 1 presents that WALA can analyze 3 versions of jQuery but it does not hold for the latest open-source version. TAJS and WALA analyze 11 and 3 versions of jQuery, respectively, but our experiments showed that they fail to analyze Mootools and Prototype unlike $\text{SAFE}_{\mathsf{lsa}}$. For the BENCH and SLICE groups, TAJS analyzes 20 and 33 programs, respectively, but WALA analyzes none of them; for the WEBSITE group, both TAJS and WALA fail to analyze any of 5 programs in the group.

We believe that $\text{SAFE}_{\mathsf{lsa}}$ is more scalable than the other analyzers because of its ability to distinguish loops more precisely without much overhead. From the experiments with jQuery, we observed that LSA distinguishes at most 4 nested loops (4-depth) across function boundaries and maximum 36 iterations (36-length). This implies that LSA can keep small numbers of loop contexts in practice even when we use big numbers for i and j in $\langle i, j, k\rangle$-LSA. Note that any combinations of the i and j values bigger than 4 and 36 in $\langle i, j, 10\rangle$-LSA give the same analysis results in the jQuery cases with the same numbers of distinguished loop contexts. Moreover, while LSA can precisely analyze loops of any forms such as `for`, `for-in`, `while`, and `do-while`, loop specialization techniques in TAJS and WALA are applicable to only special forms of loops by choosing contexts in heuristic ways; we found that Mootools and Prototype used various forms of loops, which include those that the techniques in TAJS and WALA cannot handle.

We investigated reasons why $\text{SAFE}_{\mathsf{lsa}}$ fails to analyze BootStrap[11] and some programs in the BENCH and SLICE groups within the timeout of 10 minutes. One reason is state explosion by statically indeterminate values. For example, jQuery provides the `jQuery.now()` method that returns a number representing the current time; a sound static analysis result of the

[7] `http://w3techs.com/technologies/overview/javascript_library/all`
[8] `http://www.jquery-tutorial.net/`
[9] http://www.alexa.com/topsites
[10] `http://modernizr.com`
[11] `http://getbootstrap.com/javascript/`

Table 2 Analysis results of SAFE and SAFE_{lsa} within the timeout of 5 hours.

Target	SAFE				SAFE_{lsa}			
	time (s)	MaxCALL (#)	CALL (%)	PROP (%)	time (s)	MaxCALL (#)	CALL (%)	PROP (%)
jQuery 2.1.1	timeout	32	88.57	36.36	27.56	2	99.65	83.93
Modernizr 2.8.3	60.02	49	95.42	41.67	5.92	2	96.85	100.00
BootStrap 3.3.0	timeout	37	87.10	53.33	timeout	254	87.85	73.02
Mootools 1.5.1	timeout	76	39.47	9.09	226.95	3	99.13	93.88
Prototype 1.7.2	timeout	30	91.49	28.57	35.74	2	99.41	87.72
`google.com`	timeout	22	72.22	45.45	6,433.98	10	95.22	77.56
`facebook.com`	timeout	2	99.41	42.22	timeout	3	99.60	88.14
`youtube.com`	timeout	45	94.51	42.25	3,583.09	2	99.54	89.53
`baidu.com`	timeout	37	92.00	56.52	timeout	41	93.12	72.37
`yahoo.com`	timeout	64	94.31	88.44	7,244.78	2	99.21	97.34
Average	–	39	85.45	44.39	–	32	96.95	86.34

method call should be any number. We observed that such statically indeterminate values flow into loops making analysis results of loop conditional expressions also indeterminate, which prohibits LSA from analyzing loops precisely.

Another reason we found in BootStrap is also state explosion in loops due to a sound event modeling that considers all possible event-dispatch scenarios. Event handlers for an event can access the target DOM element where the event was initially fired by the `event.target` property. Because of event bubbling and capturing [25] (event propagation through the path from the root of a DOM tree to a target element), the target element may not be `event.currentTarget` for which the current event handler has been registered. Therefore, a sound static analysis result for `event.target` should be all DOM elements on the subtree of `event.currentTarget`, and we observed that such imprecise analysis results cause state explosion by flowing into loops.

We believe that other analysis techniques not particularly related to loops may alleviate the state explosion problems. Using random constant values as in TAJS [1] or user inputs for statically indeterminate values may lessen the former problem, and more sophisticated event modeling may mitigate the latter problem. Our future work includes these directions.

6.2 Precision

For precision, we compare the analysis results of SAFE and SAFE_{lsa} for programs that just load the latest versions of the top 5 libraries and main web pages of the 5 most popular websites. To measure the analysis precision, we compute MaxCALL, CALL, and PROP that can be critical in the analysis scalability: (1) MaxCALL indicates the maximum number of possible function calls resolved during analysis for each call; bigger numbers denote more imprecise analysis results leading to more false function calls, which harm the analysis

scalability. (2) CALL indicates the ratio of definite function calls resolved to exactly one function calls to all function calls; bigger numbers denote more precise analysis results with less spurious function calls. (3) PROP indicates the ratio of dynamic property accesses resolved to constant names to all dynamic property accesses without considering direct constant accesses like `o["name"]`; bigger numbers denote more precise analysis results with more exact property accesses. Thus, the bigger MaxCALL and the smaller CALL and PROP an analysis has, the more likely it suffers from the scalability problem.

Table 2 shows the result; we averaged all figures in the table from 3 runs and normalized them to per program point to directly compare SAFE and SAFE_{lsa}; for those with timeout, we used pre-fixpoint analysis results that are still useful for precision comparison. The table shows that SAFE_{lsa} significantly improves analysis results of SAFE; it analyzes more programs than and provides more precise analysis results than SAFE. On average, LSA reduces MaxCALL from 39 to 32 and it improves CALL and PROP from 85.45% and 44.39% to 96.95% and 86.34%, respectively. In the analysis of Mootools for example, LSA dramatically improves the analysis precision by reducing MaxCALL from 76 to 3 and improving CALL and PROP from 39.47% and 9.09% to 99.13% and 93.88%, respectively. Interestingly, SAFE_{lsa} has bigger MaxCALL than SAFE for BootStrap, `facebook.com`, and `baidu.com`, which SAFE_{lsa} fails to analyze within the timeout. We found that SAFE_{lsa} reaches more program points to analyze and performs more fixpoint computation at the same program points than SAFE within the same timeout thanks to improved scalability, which increases MaxCALL at program points with imprecise analysis results. We expect that when analyzing the target programs without setting a timeout, final analysis results would show smaller MaxCALL in SAFE_{lsa} than SAFE.

6.3 Threats to Validity

A possible threat to validity of our results is that, since our target programs are simple programs that use top 5 libraries and main web pages of the 5 most popular websites, the results may not hold for some real-world JavaScript programs in different domains. Another threat is that the comparison results with other static analyzers are not independent of the capabilities of their base analyzers. As SAFE, WALA, and TAJS use different analysis techniques and different modeling of JavaScript built-in functions and DOM APIs, the better analysis results of SAFE_{lsa} may not be purely due to LSA.

7 Related Work

Sharir and Pnueli [21] introduced two approaches of call-context sensitivity, k-CFA and Summary-Based Analysis (SBA), to statically analyze functions more precisely. While k-CFA distinguishes function calls by call strings of the maximum k-length that represent call histories, SBA distinguishes function calls by input states to functions. They proved that k-CFA with the unbound length of k and SBA have the same analysis precision in domains with precision-lossless join operations. Mangal *et al.* [17] extended this result by showing that the result still holds in the presence of precision-lossy join operations. Using their notation, we formalized k-CFA and LSA in the abstract interpretation framework to prove the soundness and precision theorems of LSA. Note that although loop-sensitivity can be used with any call sensitivity techniques, LSA that we present in this work subsumes k-CFA; while k-CFA distinguishes call contexts only by some abstraction of call history denoted by call strings, LSA distinguishes loop contexts not only by abstraction of loop history but also

by abstraction of loop condition values. Thus, LSA is also similar to SBA in that it uses value abstraction for loop conditions.

Trace partitioning [10, 19] is a general theoretical framework that supports systematic abstractions on trace-based concrete semantics. While trace partitioning can improve the analysis precision of loops as well, each loop should be annotated with an unrolling number before the analysis just like the conventional loop unrolling technique. On the contrary, LSA finds precise unrolling counts for loops automatically during analysis as far as loop conditions keep determinate. Furthermore, our LSA formalization provides a detailed explanation about loop context updates in the presence of tricky program points such as `break`, `continue`, and `return` statements in loops, and our Coq mechanization proves its soundness.

SAFE [13, 15] is an extensible analysis framework for JavaScript web applications. SAFE performs sophisticated data flow analyses for JavaScript applications in a flow-sensitive and context-sensitive way producing heap information that contains pointer and value information for all variables at each program point as analysis results. In addition to the data flow analysis, it supports various extensions such as the ES6 module system [4] and detection of Web API misuses [3]. Our experiments showed that SAFE_{lsa} significantly improves the scalability and the precision of SAFE, which may, in turn, improve those of the various extensions of SAFE.

TAJS [18] is a flow and context sensitive static analyzer for JavaScript applications similar to SAFE. In addition to the object sensitivity [22], TAJS extended its context-sensitivity to distinguish more functions and loops using the values of function parameters and loop variables selected in heuristic ways, and the extension enabled TAJS to analyze most versions of jQuery [1]. However, the technique does not accompany any formalization nor soundness proof, and it is not general in that it can distinguish loops only when they conform to some specific forms; for instance, the technique does not distinguish loops where variables in loop conditional expressions are not involved in object property updates. Indeed, we found some soundness bugs and observed that TAJS cannot analyze 3 libraries among the top 5 ones.

WALA [11] is a general analysis framework originally for Java, and it analyzes JavaScript web applications as well. To address JavaScript-specific scalability problems, WALA developed the correlation tracking technique [23] that rewrites `for-in` loops in the following forms with the same property reads and writes:

```
for(x in src) des[x] = src[x];
```

to the following code:

```
for(x in src) (function (p) {des[p] = src[p]}) (x);
```

Then, the analysis distinguishes the function call in each iteration using field sensitivity (distinguishing function calls by fields of objects) [16]. Later, WALA presented the dynamic determinacy analysis [20], which first performs a dynamic information flow analysis [2] to track ever-determinate values in all concrete executions, propagates such constant values in a program, and then performs a static analysis on the specialized program. In particular for loops, the technique uses such determinate values to find the maximum unrolling numbers. However, even with the correlation tracking and dynamic determinacy techniques, WALA can analyze only 3 versions of jQuery. In contrast, LSA can apply to any forms of loops and it is fully static and automatic for finding precise numbers of distinguishable iterations for loops.

Kashyap *et al.* [14] presented JSAI, a static analysis platform to support sound analysis for JavaScript applications similar to SAFE and TAJS. One main feature of JSAI is the

configurability of sensitivity techniques including call-context sensitivity. The authors evaluated JSAI applying various call-context sensitivities such as k-CFA and object sensitivity to various benchmarks. Their experimental results showed that static analysis with higher sensitivity (as greater k for k-CFA) is far better than its counterpart with low sensitivity in terms of performance in various kinds of JavaScript programs giving more precise analysis results. It implies that higher precision may be a key to higher scalability of static analysis at least for JavaScript programs. In Section 6, We also showed that higher precision by LSA significantly improves the analysis performance for some JavaScript web applications. Because JSAI does not support the comprehensive modeling of JavaScript built-in functions including browser APIs that are essential for analyzing JavaScript web applications, we could not compare JSAI with our implementation. However, we conjecture that JSAI would fail to analyze many JavaScript web applications including JavaScript libraries unless it supports a loop specialization technique such as LSA; in Section 2, we showed that call-context sensitivity alone is not enough to analyze the most widely used JavaScript library in a scalable way due to great loss of precision in loops.

8 Conclusion

We presented a novel analysis technique, Loop-Sensitive Analysis (LSA), which distinguishes loop iterations as many as needed during analysis using loop contexts. We formalized LSA in the abstract interpretation framework and showed how to extend k-CFA to LSA. We proved that LSA is more precise than or at least as precise as k-CFA while it remains sound if the base k-CFA is sound. We provide the mechanized proofs of the soundness and precision theorems by the proof assistant tool, Coq. We have implemented LSA as an extension of an open-source JavaScript static analysis framework, SAFE. Our mechanized proofs and the LSA implementation are publicly available. We demonstrated that LSA dramatically improves the scalability of the state-of-the-art JavaScript static analyzers by enhancing analysis precision when analyzing the main web pages of the 5 most popular websites and applications that use the top 5 JavaScript libraries. Because we presented LSA as language independent, it is applicable to analysis of programs in other programming languages.

Acknowledgments. This work is supported in part by Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grants NRF-2014R1A2A2A010 03235 and NRF-2008-0062609), Samsung Electronics, and Google.

References

1. Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. In *OOPSLA'14: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2014.
2. Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS'09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. ACM, 2009.
3. SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. $\mathsf{SAFE}_{\mathsf{WAPI}}$: Web API misuse detector for web applications. In *ESEC/FSE'14: Proceedings of the 22nd ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 2014.
4. Junhee Cho and Sukyoung Ryu. JavaScript module system: Exploring the design space. In *Modularity'14: Proceedings of the 13th International Conference on Modularity*, 2014.
5. The Coq Proof Assistant. `http://coq.inria.fr/`.

6 Patrick Cousot. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications, Chapter 10*, pages 303–342. Prentice-Hall, Inc., 1981.

7 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 1977.

8 Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL'79: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 1979.

9 ECMA. ECMA-262: ECMAScript Language Specification. Edition 5.1, 2011.

10 Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Static Analysis, 5th International Symposium, SAS'98, Pisa, Italy, September 14-16, 1998, Proceedings*, pages 200–214. Springer-Verlag, 1998.

11 IBM Research. T.J. Watson Libraries for Analysis (WALA). `http://wala.sf.net`.

12 KAIST PLRG. Research material. `http://plrg.kaist.ac.kr/pch`.

13 KAIST PLRG. SAFE: Scalable analysis framework for ECMAScript. `http://safe.kaist.ac.kr`, 2014.

14 Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A static analysis platform for JavaScript. In *FSE'14: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

15 Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *FOOL'12: International Workshop on Foundations of Object Oriented Languages*, 2012.

16 Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In *CC'03: Proceedings of the 12th International Conference on Compiler Construction*. Springer-Verlag, 2003.

17 Ravi Mangal, Mayur Naik, and Hongseok Yang. A correspondence between two approaches to interprocedural analysis in the presence of join. In *ESOP 2014: Proceedings of the 23rd European Symposium on Programming*. Springer, 2014.

18 Anders Møller, Simon Holm Jensen, Peter Thiemann, Magnus Madsen, Matthias Diehn Ingesman, Peter Jonsson, and Esben Andreasen. TAJS: Type analyzer for JavaScript. `https://github.com/cs-au-dk/TAJS`, 2014.

19 Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM TOPLAS*, 29(5), 2007.

20 Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. In *PLDI'13: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2013.

21 Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications, Chapter 7*. Prentice-Hall, 1981.

22 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL'11: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2011.

23 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *ECOOP'12: Proceedings of the 26th European Conference on Object-Oriented Programming*. Springer-Verlag, 2012.

24 Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 1955.

25 W3C. Document Object Model Events. `http://www.w3.org/TR/2003/NOTE-DOM-Level-3-Events-20031107`, 2003.