35th European Conference on Object-Oriented Programming

ECOOP 2021, July 11–17, 2021, Aarhus, Denmark
(Virtual Conference)

Edited by

Anders Møller

Manu Sridharan

Editors

Anders Møller
Aarhus University, Aarhus, Denmark
amoeller@cs.au.dk

Manu Sridharan
University of California, Riverside, USA
manu@cs.ucr.edu

ACM Classification 2012
Software and its engineering

ISBN 978-3-95977-190-0

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-190-0.

Publication date
July, 2021

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 https://portal.dnb.de.

Digital Object Identifier: 10.4230/LIPIcs.ECOOP.2021.0

ISBN 978-3-95977-190-0 **ISSN 1868-8969** **https://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.

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Regular Papers

35th European Conference on Object-Oriented Programming (ECOOP 2021).
Editors: Manu Sridharan and Anders Møller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Pearls

Message from the Chairs

It is our great pleasure to welcome you to ECOOP 2021, to be held during July 11–17. ECOOP is Europe's longest-standing annual Programming Languages conference, bringing together researchers, practitioners, and students to share their ideas and experiences in all topics related to programming languages, software development, object-oriented technologies, systems and applications.

ECOOP 2021 was originally planned to take place at Aarhus University, Denmark, but the COVID-19 pandemic made that impossible, so again this year it will be a virtual conference. As well as technical papers and keynotes, ECOOP 2021 features a doctoral symposium, a poster session, and a summer school. The event is co-located with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), the International SPIN Symposium on Model Checking of Software, the Rebase conference, and the following workshops:

- Workshop on AI and Software Testing/Analysis (AISTA)
- International Workshop on Smart Contract Analysis (WOSCA)
- International Workshop on Verification of Objects at Runtime Execution (VORTEX)
- International Workshop on Context-Oriented Programming and Advanced Modularity (COP)
- Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems (ICOOOLPS)
- Workshop on Formal Techniques for Java-like Programs (FTfJP)

The ECOOP/ISSTA Summer School consists of invited lectures by Eric Bodden, Marcel Böhme, Claire Le Goues, Satish Chandra, and Andreas Rossberg. The summer school, which was organized by Frank Tip and Andreas Zeller, aims to provide undergraduate and graduate students and postdocs with a gentle introduction to research that is being conducted in the ECOOP and ISSTA communities. The ECOOP/ISSTA Doctoral Symposium, organized by Wei Le and Eric Bodden, provides a forum for PhD students at any stage in their research to get detailed feedback and advice, and to establish new research collaborations.

To make it easy for our international community to attend the event from anywhere across the globe, ECOOP/ISSTA will use a 3-time-band format, with each paper presentation being given twice (in the two time bands that are most convenient for the speaker) so that all attendees are able to attend most talks at reasonable times.

Paper selection process

As in recent years, ECOOP 2021 supported a "journal first" track in addition to the traditional approach of direct paper submission to be considered for the proceedings. The *Science of Computer Programming* ECOOP 2021 Special Issue contains one paper that will be presented at the conference, and a single paper from *ACM Transactions on Programming Languages and Systems* will also be presented. For traditional submissions, ECOOP 2021 again supported the six paper categories introduced in ECOOP 2019: Research Paper, Tool Insight Paper, Reproduction Study, Experience Report, Pearl, and Brave New Idea.

35th European Conference on Object-Oriented Programming (ECOOP 2021).
Editors: Manu Sridharan and Anders Møller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In total, ECOOP 2021 received 57 submissions, of which 22 were accepted (38.6% acceptance rate). There were 50 Research Paper submissions (20 accepted), 3 Pearls (2 accepted), 2 Tool Insight submissions (none accepted), 1 Experience Report (none accepted), and 1 Brave New Idea (none accepted). For the *Science of Computer Programming* Special Issue there were 3 submissions, with 1 accepted.

ECOOP 2021 had a single Program Committee with 32 members, and no External Review Committee. ECOOP 2021 used a *strong* double-blind review process: author identities of accepted papers were only revealed after decisions were made, and author identities for rejected papers were never revealed. Further, the review process took place fully online, with no in-person PC meeting. Given the fully online review process and strong double-blind reviewing, submissions from PC members could be handled via standard conflict mechanisms, and there was no need for a separate external review committee.

Each submission was reviewed by at least three members of the Program Committee and selected additional reviewers. Authors were given a chance to respond to all reviews of their paper, except in rare cases where an additional review was solicited after the author response period. For most papers, particularly those where further discussion occurred after the author response, the authors were provided with a summary of the reviewer discussion, and in the case of a reject decision, the main reasons for the rejection.

New for ECOOP 2021, all authors were asked about their *intent to submit an artifact* at the point of paper submission. Authors could indicate whether they intended to submit an artifact if their paper was accepted, and optionally give an explanation if no artifact would be submitted. It was made clear to authors that artifacts may not be appropriate for all papers and were *not* required. All 20 papers accepted in the Research Paper category had expressed an intent to submit an artifact, and in the end 19 artifacts were submitted for evaluation (the authors of the last paper could not make the artifact evaluation deadline due to illness, but they still made a public artifact). Asking for artifact intent during paper submission successfully led to a very high submission rate for artifact evaluation, and we hope future ECOOP chairs will continue to use this process.

Acknowledgements

Organizing ECOOP has involved many members of our community, and we would like to express our gratitude towards all the people involved. We are particularly thankful to Cristian Cadar, the General Chair of ISSTA 2021, for enjoyable and effective collaboration about the organization of the joint events, and to Lucie Lerch for managing finances and coordination wiht AITO. We thank the Program Committee members and external reviewers for the thorough reviews and discussions of the submitted papers, and the Artifact Evaluation Committee for their efforts.

Many other people contributed to various aspects of the conference. We thank Marcel Böhme and Maria Christakis for organizing an exciting collection of workshops, Ajitha Rajan and Sebastian Erdweg for managing the poster sessions, Omer Tripp and Darko Marinov for successfully attracting corporate supporters, and Lisa Nguyen Quang Do for taking care of publicity. We also thank Daniel Grumberg for assisting with the website and video upload system and Elmer van Chastelet for providing excellent support and accommodating our requests for new features in the conf.researchr.org system.

We gratefully acknowledge our sponsor AITO and our financial supporters, Google, Dragon Testing, Amazon, Microsoft Research, KBR and NASA, Facebook, and JetBrains, as well as the cooperation with ACM and SIGPLAN. Thanks to the generous contributions from the financial supporters, participation at ECOOP 2021 and the affiliated events is free.

Finally, we want to thank all the authors for submitting their work and the attendees for contributing to making the conference a success. We hope that you will find the ECOOP 2021 program inspiring and valuable, and that the conference will bring new ideas and give opportunities to meet with researchers and practitioners in our community.

Anders Møller
ECOOP 2021 General Chair
Aarhus University

Manu Sridharan
ECOOP 2021 Program Chair
University of California, Riverside

Message from the Artifact Evaluation Chairs

The goals of the *Artifact Evaluation* (AE) are to foster the reproducibility of results by providing authors the possibility to submit an artifact for accepted papers. Artifacts include, but are not limited to, software artifacts, data sets, and proofs. An *Artifact Evaluation Committee* (AEC) reviews these artifacts and decides upon their acceptance. The accepted artifacts are archived in the *Dagstuhl Artifacts Series* (DARTS) published on the *Dagstuhl Research Online Publication Server* (DROPS). Each artifact is assigned a *Digital Object Identifier* (DOI) that can be used in future citations.

This year, the committee evaluated 19 artifacts out of 20 papers accepted at the conference's research track. This corresponds to a record participation rate of 95%. 15 of those artifacts were accepted (a 79% acceptance rate). In total, 75% of the regular research papers published at ECOOP 2021 have successfully passed the AE process, indicated by an artifact-evaluation badge on the paper. The improvement from last year continues: from 2017 to 2020, respectively 59%, 38%, 50%, and 70% of the research papers were accompanied by accepted artifacts.

The AE process for 2021 was a continuation of the AE process of previous ECOOP editions. In particular, the process was still based on the artifact evaluation guidelines by Shriram Krishnamurthi, Matthias Hauswirth, Steve Blackburn, and Jan Vitek published on the Artifact Evaluation site. The guidelines for artifacts that contain mechanized proofs developed by the ECOOP 2018 AEC were also reused to help both reviewers and authors in creating and reviewing such artifacts.

Each artifact was evaluated by two AEC members, which corresponded to a reviewer load of two artifacts. The reviewing process consisted of three phases.

- In the "kick-the-tires" phase, reviewers briefly verified the basic integrity of the artifacts to discover any issues that could prevent the evaluation of the artifact (e.g., a corrupted virtual machine image) and to assign a grade for the getting-started guide.
- In case of any issues, reviewers could, as part of a response phase, indicate issues and ask clarifying questions to the authors. Authors could respond to the reviewers' first feedback, and update their artifacts to address any issues that were raised by the reviewers.
- In the main review phase, each reviewer had two weeks to do a comprehensive evaluation of each artifact. Reviewers were asked to assess the consistency of the artifact with respect to the paper, the artifact's completeness, documentation, and reusability for future research and to decide on an overall grade. The review phase was followed by a discussion phase, in which artifacts were discussed to converge on either the artifacts' acceptance or rejection. Authors that received an acceptance notification were given two weeks to incorporate reviewers' feedback and submit the camera-ready version of their artifacts.

We would like to thank the 22 members of this year's AEC, who donated their valuable time and effort to make the AE process possible. We would also like to thank Michael Wagner for the publication of the artifacts volume, as well as ECOOP 2021's General Chair Anders Møller and the Program Chair Manu Sridharan for helping us coordinate the artifact evaluation with the paper review process.

William G.J. Halfond
Artifact Evaluation Co-Chair
University of Southern California

Quentin Stiévenart
Artifact Evaluation Co-Chair
Vrije Universiteit Brussel

35th European Conference on Object-Oriented Programming (ECOOP 2021).
Editors: Manu Sridharan and Anders Møller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Foreword by the President of AITO

Dear ECOOP participants,

It has been yet another year dominated by COVID-19 and so ECOOP again will be virtual – last year was successful and this year is looking even better: The organizers have done a great job and the program is exciting. On the up-side of doing the event virtually is that even more will be able to attend.

ECOOP 2021 is co-located (in the virutal world) with ISSTA 2021 – a cooperation that traditionally is fruitful.

I would like to thank the organizers – lead by Anders Møller – and the PC – lead by PC Chair Manu Sridharan – for their parts in making ECOOP successful – and the authors for their contributions – they provide the essentials that we are meeting to discuss and learn from.

May ECOOP be a good experience for you – and let's hope that for ECOOP 2022, we again will be able to meet physically and enjoy both a great scientific program and the benefits of social interaction.

All the best,

Eric Jul
AITO President

35th European Conference on Object-Oriented Programming (ECOOP 2021).
Editors: Manu Sridharan and Anders Møller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Organization

General Chair

Anders Møller *(Aarhus University, Denmark)*

Program Chair

Manu Sridharan *(University of California at Riverside, USA)*

Artifact Evaluation Co-Chairs

William G.J. Halfond *(University of Southern California, USA)*
Quentin Stiévenart *(Vrije Universiteit Brussel, Belgium)*

Workshop Co-Chairs

Marcel Böhme *(Monash University, Australia)*
Maria Christakis *(Max Planck Institute for Software Systems, Germany)*

Doctoral Symposium Co-Chairs

Wei Le *(Iowa State University, USA)*
Eric Bodden *(Paderborn University and Fraunhofer IEM, Germany)*

Summer School Co-Chairs

Frank Tip *(Northeastern University, USA)*
Andreas Zeller *(CISPA Helmholtz Center for Information Security, Germany)*

Posters Co-Chairs

Ajitha Rajan *(University of Edinburgh, UK)*
Sebastian Erdweg *(Johannes Gutenberg University Mainz, Germany)*

Sponsorship Co-Chairs

Omer Tripp *(Amazon, USA)*
Darko Marinov *(University of Illinois at Urbana-Champaign, USA)*

Finance Chair

Lucie Lerch *(Czech Technical University, Czech Republic)*

Publicity Chair

Lisa Nguyen Quang Do *(Google, Switzerland)*

35th European Conference on Object-Oriented Programming (ECOOP 2021).
Editors: Manu Sridharan and Anders Møller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Program Committee

Alexander J. Summers *(University of British Columbia, Canada)*
Alexandra Silva *(University College London, United Kingdom)*
Burcu Kulahcioglu Ozkan *(Delft University of Technology, Netherlands)*
Camil Demetrescu *(Sapienza University of Rome, Italy)*
Colin Gordon *(Drexel University, USA)*
David Grove *(IBM Research, USA)*
Eelco Visser *(Delft University of Technology, Netherlands)*
Elena Zucca *(University of Genova, Italy)*
Eric Bodden *(Paderborn University and Fraunhofer IEM, Germany)*
Fernando Magno Quintão Pereira *(Universidade Federal de Minas Gerais, Brazil)*
George Fourtounis *(University of Athens, Greece)*
Hakjoo Oh *(Korea University, South Korea)*
Hila Peleg *(University of California, San Diego, USA)*
Jens Dietrich *(Victoria University of Wellington, New Zealand)*
John Wickerson *(Imperial College London, United Kingdom)*
Jonathan Aldrich *(Carnegie Mellon University, USA)*
Julia Lawall *(Inria, France)*
Lingming Zhang *(University of Illinois Urbana-Champaign, USA)*
Lu Zhang *(Peking University, China)*
Michael Greenberg *(Pomona College, USA)*
Michael Pradel *(University of Stuttgart, Germany)*
Mira Mezini *(TU Darmstadt, Germany)*
Murali Krishna Ramanathan *(Uber Technologies Inc., USA)*
Omer Tripp *(Amazon Inc., USA)*
Robert O'Callahan *(Pernosco, New Zealand)*
Sam Tobin-Hochstadt *(Indiana University, USA)*
Sukyoung Ryu *(KAIST, South Korea)*
Todd Mytkowicz *(Microsoft, USA)*
Uday Khedker *(IIT Bombay, India)*
Viktor Kunčak *(EPFL, Switzerland)*
Walter Binder *(University of Lugano, Switzerland)*
Werner Dietl *(University of Waterloo, Canada)*

Artifact Evaluation Committee

Ali Shokri *(Rochester Institute of Technology, USA)*
Anil Koyuncu *(University of Luxembourg, Luxembourg)*
Arnab Sharma *(Paderborn University, Germany)*
Asmae Heydari Tabar *(TU Darmstadt, Germany)*
Chaitanya Koparkar *(Indiana University, USA)*
Chengyu Zhang *(East China Normal University, China)*
Crystal Chang Din *(University of Oslo, Norway)*
Eduard Kamburjan *(University of Oslo, Norway)*
Giovanni Ciatto *(Università di Bologna, Italy)*
Jordan Samhi *(University of Luxembourg, Luxembourg)*
Junwen Yang *(University of Chicago, USA)*
Jyoti Prakash *(National University of Singapore, Singapore)*

Krishna Narasimhan *(TU Darmstadt, Germany)*
Lorenzo Testa *(Università degli Studi di Torino, Italy)*
Narges Shadab *(University of California at Riverside, USA)*
Pietro Barbieri *(Università degli Studi di Genova, Italy)*
Pinjia He *(ETH Zurich, Switzerland)*
Raphaël Monat *(LIP6, Sorbonne Université, France)*
Shukun Tokas *(SINTEF Digital, Oslo, Norway)*
Somesh Singh *(Indian Institute of Technology Madras, India)*
Utpal Bora *(Institute of Technology Hyderabad, India)*
Yusuke Izawa *(Tokyo Institute of Technology, Japan)*

External Reviewers

Davide Ancona *(Universita di Genova, Italy)*
Matija Pretnar *(University of Ljubljana, Slovenia)*

35th European Conference on Object-Oriented Programming (ECOOP 2021).
Editors: Manu Sridharan and Anders Møller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Authors

Jonathan Aldrich (3)
Carnegie Mellon University,
Pittsburgh, PA, USA

Tomoyuki Aotani (17)
Mamezou Co.,Ltd., Tokyo, Japan

Sven Apel (14)
Saarland University, Saarland Informatics
Campus, 66123 Saarbrücken, Germany

Ellen Arteca (7)
Northeastern University, Boston, MA, USA

Johannes Bader (3)
Jane Street, New York, NY, USA

Christian Bartolo Burlò (20)
Gran Sasso Science Institute, L'Aquila, Italy

Raef Bassily (8)
Ohio State University, Columbus, OH, USA

Jan Bessai (4)
Technische Universität Dortmund, Germany

Eric Bodden (2, 19)
Heinz Nixdorf Institute, Paderborn, Germany;
Paderborn University, Germany;
Fraunhofer IEM, Paderborn, Germany

Rodrigo Bonifácio (19)
Computer Science Department,
University of Brasília, Brazil

Edwin Brady (9)
School of Computer Science,
University of St Andrews, Scotland, UK

Maxime Buyse (5)
Uber Elevate, Paris, France

Ornela Dardha (10)
School of Computing Science,
University of Glasgow, Scotland, UK

Rémi Delmas (5)
Uber Elevate, Paris, France

Aleksandar S. Dimovski (14)
Mother Teresa University, Skopje,
North Macedonia

Boris Düdder (4)
University of Copenhagen, Denmark

Sam Estep (3)
Carnegie Mellon University,
Pittsburgh, PA, USA

Simon Fowler (10)
School of Computing Science,
University of Glasgow, Scotland, UK

Adrian Francalanza (20)
Department of Computer Science,
University of Malta, Msida, Malta

Yaoqing Gao (16)
Huawei, Toronto, Canada

Simon J. Gay (10)
School of Computing Science,
University of Glasgow, Scotland, UK

Saverio Giallorenzo (22)
Università di Bologna, Italy; INRIA, Sophia
Antipolis, France

Xiaodong Gu (11)
School of Software, Shanghai Jiao Tong
University, China

Youssef Hamadi (5)
Uber Elevate, Paris, France

Yo-Sub Han (11)
Department of Computer Science,
Yonsei University, Seoul, South Korea

Yu Hao (8)
Ohio State University, Columbus, OH, USA

Paul Harvey (10)
Rakuten Mobile Innovation Studio,
Tokyo, Japan

Dongjie He (16)
University of New South Wales,
Sydney, Australia

George T. Heineman (4)
Worcester Polytechnic Institute, MA, USA

Ben Hermann (2)
Technische Universität Dortmund, Germany

Xuejing Huang (12)
The University of Hong Kong, Hong Kong

Ranjit Jhala (18)
University of California, San Diego, CA, USA

Tetsuo Kamina (17)
Oita University, Japan

35th European Conference on Object-Oriented Programming (ECOOP 2021).
Editors: Manu Sridharan and Anders Møller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Sunghun Kim (11)
Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong

Matthew Kolosick (18)
University of California, San Diego, CA, USA

Alexander Kovalenko (6)
Czech Technical University in Prague, Czech Republic

Stefan Krüger (19)
Independent Researcher, Munich, Germany

Sufian Latif (8)
Ohio State University, Columbus, OH, USA

Jingbo Lu (16)
University of New South Wales, Sydney, Australia

Petr Maj (6)
Czech Technical University in Prague, Czech Republic

Hidehiko Masuhara (17)
Tokyo Institute of Technology, Japan

Mira Mezini (19)
Technical University of Darmstadt, Germany

Fabrizio Montesi (22)
University of Southern Denmark, Odense, Denmark

Krishna Narasimhan (19)
Technical University of Darmstadt, Germany

Bruno C. d. S. Oliveira (12)
The University of Hong Kong, Hong Kong

Marco Peressotti (22)
University of Southern Denmark, Odense, Denmark

Ohad Rau (13)
Georgia Institute of Technology, Atlanta, GA, USA

David Richter (22)
Technical University of Darmstadt, Germany

Atanas Rountev (8)
Ohio State University, Columbus, OH, USA

Guido Salvaneschi (22)
University of St. Gallen, Switzerland

Vivek Sarkar (13)
Georgia Institute of Technology, Atlanta, GA, USA

Alceste Scalas (20)
DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark

Philipp Dominik Schubert (2)
Heinz Nixdorf Institute, Paderborn, Germany

Max Schäfer (7)
GitHub, Oxford, UK

Marco Servetto (21)
ECS, Victoria University of Wellington, New Zealand

Xipeng Shen (15)
Department of Computer Science, North Carolina State University, Raleigh, NC, USA

Konrad Siek (6)
Czech Technical University in Prague, Czech Republic

Joshua Sunshine (3)
Carnegie Mellon University, Pittsburgh, PA, USA

Éric Tanter (3)
Computer Science Department (DCC), University of Chile, Santiago, Chile

Frank Tip (7)
Northeastern University, Boston, MA, USA

Anish Tondwalkar (18)
University of California, San Diego, CA, USA

Hendrik van Antwerpen (1)
Delft University of Technology, The Netherlands

Eelco Visser (1)
Delft University of Technology, The Netherlands

Jan Vitek (6)
Czech Technical University in Prague, Czech Republic; Northeastern University, Boston, MA, USA

Caleb Voss (13)
Georgia Institute of Technology, Atlanta, GA, USA

Pascal Weisenburger (22)
University of St. Gallen, Switzerland

Jenna Wise (3)
Carnegie Mellon University, Pittsburgh, PA, USA

Jingling Xue (16)
University of New South Wales, Sydney, Australia

Wenjia Ye (12)
The University of Hong Kong, Hong Kong

Guoqiang Zhang (15)
Department of Computer Science,
North Carolina State University,
Raleigh, NC, USA

Hailong Zhang (8)
Fordham University, New York, NY, USA

Hongyu Zhang (11)
The University of New Castle, Australia


Elena Zucca (21)
DIBRIS, University of Genova, Italy

Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers

Hendrik van Antwerpen
Delft University of Technology, The Netherlands

Eelco Visser
Delft University of Technology, The Netherlands

Abstract

Compilers that can type check compilation units in parallel can make more efficient use of multi-core architectures, which are nowadays widespread. Developing parallel type checker implementations is complicated by the need to handle concurrency and synchronization of parallel compilation units. Dependencies between compilation units are induced by name resolution, and a parallel type checker needs to ensure that units have defined all relevant names before other units do a lookup. Mutually recursive references and implicitly discovered dependencies between compilation units preclude determining a static compilation order for many programming languages.

In this paper, we present a new framework for implementing hierarchical type checkers that provides implicit parallel execution in the presence of dynamic and mutual dependencies between compilation units. The resulting type checkers can be written without explicit handling of communication or synchronization between different compilation units. We achieve this by providing type checkers with an API for name resolution based on scope graphs, a language-independent formalism that supports a wide range of binding patterns. We introduce the notion of *scope state* to ensure safe name resolution. Scope state tracks the completeness of a scope, and is used to decide whether a scope graph query between compilation units must be delayed. Our framework is implemented in Java using the actor paradigm. We evaluated our approach by parallelizing the solver for Statix, a meta-language for type checkers based on scope graphs, using our framework. This parallelizes every Statix-based type checker, provided its specification follows a split declaration-type style. Benchmarks show that the approach results in speedups for the parallel Statix solver of up to 5.0x on 8 cores for real-world code bases.

2012 ACM Subject Classification Software and its engineering → Compilers; Theory of computation → Parallel algorithms

Keywords and phrases type checking, name resolution, parallel algorithms

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.1

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.1`

Funding NWO VICI Language Designer's Workbench project (639.023.206).

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

Despite the general availability of multi-core architectures, many compilers do not take advantage of these for type checking. Parallelizing a compiler remains a challenging task, which requires dealing with explicit synchronization and communication between compilation units. For example, the authors of GCC made the following remark about their efforts to parallelize parts of the compiler [6]:

> *"One of the most tedious parts of the job was [...] making several global variables threadsafe, and they were the cause of most crashes in this project."*

They continue to say that even with the help of specialized tools it remained difficult to do the parallelization correctly. Compilers for many major languages do not support parallel front ends, or only experimentally. Some build tools allow the compiler to be run in parallel, but many require a static compilation order, and because they have no internal knowledge of the language being compiled, cannot generically handle cyclic dependencies between compilation units. A generic, reusable solution to the problem of how to implement type checkers that can process compilation units in parallel, which correctly deals with (potentially cyclic) dependencies between units, is missing[1].

Dependencies between compilation units are the results of name lookup from one unit into another. A correct concurrent type checker must ensure that when a lookup is done, all relevant units have progressed enough to provide a complete answer. For languages that support true separate compilation (e.g., [23]), there are no lookups into other units, and processing them in parallel is trivial. It may also be possible to run type checkers in parallel using a compilation order based on static or dynamic dependencies, which ensures units are compiled after their dependencies. But many programming languages have features, such as mutually recursive modules, that result in mutual dependencies between compilation units. When compilation units are mutually dependent, neither unit can be completed before the other is at least partially checked. A more fine-grained approach than processing compilation units in a fixed order is required.

This paper presents a new framework for the implementation of type checkers that provides *implicit parallel execution*. Type checkers are organized as a hierarchy of compilation units, which allows modeling simple scenarios such as flat files in a project, as well as package hierarchies. The framework supports *dynamic dependencies* and *mutual dependencies* between compilation units. The type checkers can be written without the need to explicitly handle communication or synchronization between units.

This is achieved by providing type checkers with an API for name resolution based on scope graphs. Scope graphs are a language-independent formalism for name binding and name resolution, which has been shown to support a wide range of binding patterns, and has successfully been applied to implement type checkers [16, 31, 20]. The key to our approach is twofold:

- delay lookups when other units have not progressed enough to give a *safe*, that is, complete answer, and
- release delayed queries as soon as possible, even if other parts of the graph are still incomplete.

Recent work by Rouvoet et al. [20] identifies the absence of *weakly critical edges* as a sufficient condition to guarantee safe name resolution in a partial scope graph. We develop the notion of *scope state* to allow fine-grained tracking of the presence or absence of weakly critical edges. Through these scope states, which are managed by the type checkers via the name resolution API, the framework ensures safety of name resolution. The provided API is asynchronous, which works with type checkers that follow a synchronous pattern, where every name resolution query is awaited, as well as with type checkers that use dynamic scheduling techniques, such as worklists and continuations.

[1] The type checkers we envision are both concurrent (i.e., units make (interleaved) progress during the same period) and parallel (i.e., units run at the same time), and we use the terms interchangeably.

```
package p;

class A {
    p.C f;
    static class C {}
}
```

```
package p;

class B extends A {
    C g;
}
```

```
package p;

class C {
    A h;
}
```

Figure 1 Three unit Java program demonstrating mutual and discovered dependencies.

We claim the following contributions:

- We propose the notion of *scope state* to explicitly track the presence of weakly-critical edges (Section 3.3).
- We introduce a model of hierarchical compilation units with scope sharing (Section 4.1). We extend scope state with a notion of sharing, which allows us to track weakly-critical edges in the hierarchy of compilation units (Section 4.2).
- We present a scope graph-based name resolution API for use by type checker implementations (Section 4.3).
- We present an actor-based algorithm that implements the hierarchical compilation unit model and the name resolution API, and provides implicit parallel execution of the compilation units (Section 5).
- We present a fine-grained deadlock handling approach to ensure termination that is well-suited for interactive applications of the type checkers (Section 5).
- We show that our framework captures the scheduling behavior of Rouvoet et al. [20] by porting the Statix solver to our framework. We discuss local inference and the need for a specification style that models declarations and their types as separate scopes (Section 6). We parallelize all Statix type checkers, provided they follow this specification style.
- We benchmark the parallel Statix solver using a specification for a subset of Java on a few real world projects, showing speedups up to 5.0x on 8 cores for larger projects.

All the source code and benchmark results are available in the accompanying artifact.

2 Motivation and Scope

Our goal is to develop a framework that provides implicit parallelization of type checkers. In this section we discuss the features we want to support, the difficulties these features pose to parallelization, and an overview of our solution.

We use an example of a Java program consisting of three compilation units, shown in Figure 1, to illustrate the requirements on parallel type checkers. This example shows to two dependency patterns that are challenging for parallelization. The first is *mutual dependencies*. Class `A` refers to class `p.C` (qualified to distinguish it from the nested class it defines), while class `C` refers to class `A`. The second is *dynamic dependencies*. These are dependencies that are discovered during type checking, and that are not obvious without at least partially checking the program. The reference from `B` to `C` is an example of this. The name `C` could refer to the top-level class in the package, or to the nested class in `A`. To decide that it refers to the nested class in `A`, we need to resolve the reference to the super class `A` and its interface.

Typical compiler design (e.g., [2]) divides compilation into phases, including parsing, type checking, and code generation. We focus on the type checking phase, which is often difficult to parallelize because of the context dependence of type checking and name resolution. The type checking phase of the compiler for our example may consist of several steps: (a) build a

symbol table containing information on defined classes and type inheritance, (b) build the interface for each class by processing field and method declarations, and (c) check the field and method bodies of each class. Each step depends on the information collected in the previous phases.

What does it take to parallelize this type checker? Compilation units are checked in parallel, but inevitably need information provided by other compilation units. Mutual dependencies between compilation units prevents linear ordering of compilation units, while dynamic dependencies mean part of the work must be done before all dependencies are even known. This immediately rules out simple parallelization schemes based on a topological ordering of compilation units, where units are checked after their dependencies have finished.

The main challenge introduced by parallelization is therefore how to deal with partial information during type checking (which has been called the Doesn't Know Yet Problem [22]). For example, the compilation unit for class `B` does a lookup of nested classes in its super class `A`, while the compilation unit for `A` has not constructed its interface yet. Solving this problem may require designing locking schemes for threads with shared data, or messaging protocols for communicating processes, as well as keeping track of the completeness of (part of) the symbol table or interface. Designing concurrent software is notoriously hard, and bugs can result in deadlocks or invalid results because of the use of incomplete information.

Our solution to this problem is a framework that allows compiler engineers to write their type checker without concern for parallelization. All the work of coordinating the parallel units and keeping track of completeness of interfaces is handled by the framework. The key idea is that all dependencies between units are the result of either a hierarchy between units (e.g., compilation units in packages) or name[2] lookups. The framework provides the type checker with a name handling API based on the expressive name binding model of scope graphs. Scope graphs are language-independent, and have been successfully used to model a wide variety of binding patterns, including mutually recursive modules, type-dependent names, and generics. Type checkers use this API to declare the name binding and scoping structure, and resolve names by queries on the resulting scope graph. The scope graphs grows monotonically with the type checker marking the parts of the graph that are completed. In return, the framework completely hides the communication between different compilation units and ensures only complete information is used to answer name resolution queries.

In terms of our earlier example, this means that the type checker of each compilation unit follows the steps of the original, non-parallel, design. When the type checker for class `B` queries the not-yet-constructed interface of class `A`, the query is simply suspended until the unit of `A` has progressed enough to be able to answer the query. The type checker of unit of `A`, on the other hand, is unaware of the query as the delay and answer mechanism is handled transparently based on the monotone completion of `A`'s scope graph.

3 Type Checking with Scope Graphs

Scope graphs [16] are a language-independent formalism to specify name binding and name resolution, and a key ingredient of our approach. In this section we explain scope graphs, describe the problem of safe name resolution and its solution using critical edges [20], and we introduce the notion of scope state to track the presence of critical edges.

[2] We use name in a broad sense, as it can be complex data and does not necessarily have to appear literally in the original program.

3.1 Scope Graphs

Scope graphs describe the name binding structure of programs as a directed graph of scopes with associated data, connected by labeled edges. Scopes correspond to regions in the program that behave uniformly with respect to name binding. Name resolution corresponds to queries in the graph that find paths from references to scopes with matching associated data (e.g., an identifier name).

The program in Figure 2 will be our running example. It consists of two files, one containing a class `A` in a package `p`, and a class `B` in a package `q`. Class `B` extends class `A` and refers to a field `f` in `A` from the method `m`. The scope graph corresponding to our example is shown in Figure 3.[3] The node labeled s_R represents the root scope of the program. The scopes s_p, s_q, s_A, s_B, s_f, and s_m correspond to declarations in the program, and each has the simple class name as associated data, depicted as $s \to x$. Edges from the containing scopes to these declarations are labeled to indicate the kind of declaration: PKG, CLS, FLD and MTHD for package, class, field, and method declarations respectively. The scopes $s_{T(f)}$ and $s_T(m)$ represent the types of the declarations `f` and `m`, and each is connected to their declaration with a TYPE-labeled edge. The label LEX is used for connecting a scope to its lexical parent.[4] Finally, class extension is modeled by an EXT-labeled edge between the two class scopes, which makes the declarations from the super-class reachable from the subclass.

Name resolution is expressed by means of queries over the scope graph, finding a path from a reference's scope to a matching declaration. Query parameters control *reachability* and *disambiguation*. What data in the graph is reachable is specified with a regular expression describing valid paths and a predicate describing matching data. For example, to resolve a reference `f` in the lexical context or in a super-class, one would use the regular expression $\mathsf{LEX}^*\mathsf{EXT}^*\mathsf{FLD}$, and a predicate matching the name `f`. Disambiguation determines which data is visible if multiple reachable results are found, and is specified with an order on edge labels and a predicate describing equivalent data. For example, if we prefer local definitions of field references over definitions that traverse more edges, and definitions found in super-classes over definitions found in the lexical context, we would use a label order $\$ < \mathsf{EXT} < \mathsf{LEX}$. The special label \$, assumed different from all user-provided labels, is used for end-of-path.

We use the following notation: A scope graph $\mathcal{G}$ is a triple $\langle S, E, \rho \rangle$ of scope identifiers S, edges E, and a partial function ρ from scopes to associated data. The function $\mathit{scopes}(\mathcal{G})$, $\mathit{edges}(\mathcal{G})$, and $\mathit{data}(\mathcal{G})$ project the three components of the triple. Query parameters are a path well-formedness regular expression re, a data matching predicate $\mathbf{D}$, and a strict partial order on labels $\mathbf{L}$. We use $\mathbf{D}_{\mathtt{x}}$ for the predicate matching the name `x`. The result of a query is an answer set A of tuples (p, d) of a path p and a datum term d. A path p is either a single scope s, or a labeled step $p \cdot l \cdot s$, and $\mathit{target}(p)$ projects the last scope of path.

[3] This scope graph is a simplification of the scope graph that would be necessary to support all Java's name resolution features, and does not account for the possibility of named and wildcard imports, implicit package visibility, nested classes, etc.

[4] We say binding is *lexical* if binders are only visible in sub-terms of the term where they are bound. Examples are lambda and let expressions. If the scope of the binder is wider, we say the binding is *non-lexical*. Examples are identifiers imported from modules, or references to members on expressions of a class/record type.

```
package p;
public class A {
    int f;
}
```

```
package q;
public class B extends p.A {
    public int m() {
        return f;
    }
}
```

Figure 2 Example Java program with two compilation units.

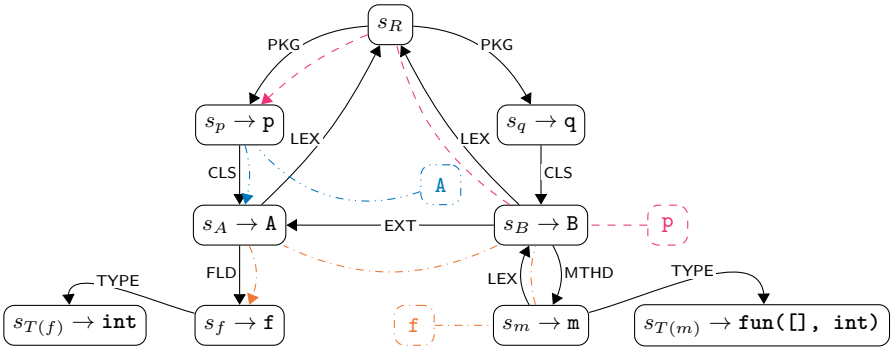


Figure 3 Scope graph corresponding to the Java program in Figure 2.

3.2 Critical Edges for Safe Name Resolution

Type checkers use the scope graph to resolve names, but must also construct the scope graph. When type checking starts, all we have is an empty scope graph, which is gradually built up as type checking progresses. Rouvoet et al. [20] observe that it is not always possible to construct the full scope graph without already querying it. This can be seen in our example in Figure 3 as well. The construction of the extension edge from s_B to s_A requires resolving the reference to `A` in a partial graph. This raises the question when it is safe to do so. After all, if that query was executed before the declaration of `A` is added to the graph, it results in an undeserved error. A query is considered *safe* to resolve when its result in the current partial scope graph is the same as its answer in the final scope graph. Rouvoet et al. [20] identify the absence of *critical edges* as the condition to guarantee safety. A critical edge is the first edge that is missing in the partial graph that will be part of the query result in the final graph. For example, if the graph in Figure 3 was complete except for the EXT edge between s_B and s_A, this edge is critical for the resolution of the reference `f`. But the same missing edge is not critical for the resolution of `p`.

Since determining the critical edges in an incomplete scope graph amounts to solving the whole name resolution problem, Rouvoet et al. [20] propose *weakly critical edges* as a conservative approximation of critical edges. Weakly critical edges are missing edges that *may* lead to a result for the query. In our earlier example, the missing EXT edge is weakly critical for the resolution of `f` even if s_A does not eventually contain a declaration for `f`.

3.3 Scope States

The final step to guarantee safe resolution is then to determine the weakly critical edges. The solution of Rouvoet et al. [20] is a predicate over a constraint set, which is specific to the Statix meta-language. The crucial property of their safety predicate is that the set of weakly critical edges only decreases as type checking progresses. They prove that this ensures

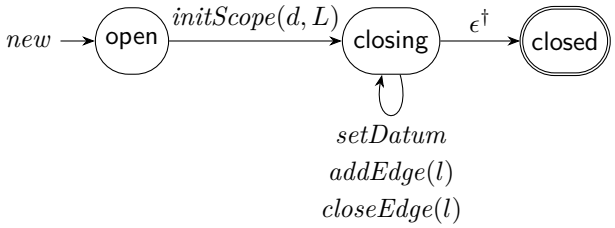


Figure 4 Scope states. Transition diagram.

$\{\top\}$	*new*	$\{O = \emptyset\}$
$\{\top\}$	$initScope(d, L)$	$\{O = L \uplus \{\$ \mid d = \top\}\}$
$\{\$ \in O\}$	*setDatum*	$\{\$ \notin O\}$
$\{l \in O\}$	$addEdge(l)$	$\{\top\}$
$\{l \in O\}$	$closeEdge(l)$	$\{l \notin O\}$
$\{O = \emptyset\}$	$\epsilon^\dagger$	$\{\top\}$

Figure 5 Scope states. Transition conditions and effects.

that once a query is executed, there can not be additions to the scope graph that lead to new results for that query. Our purpose is to develop a language-independent framework for parallel type checkers that correctly handles the dependencies between compilation units. Dependencies are the result of name resolution, thus handling the dependencies between units means ensuring name resolution between units is correct. If we can capture the presence of weakly critical edges independently of the particular type checker and object language, we can provide a general mechanism to delay queries until they are safe to execute.

To that purpose, we introduce the notion of *scope state*, which consists of a *state* (open, closing, and closed) and a set of *open labels* O, consisting of edge labels l or the special data label \$. Then, weakly critical edges are characterized by scopes that are open and/or have open edges. When a scope is closed, it is always safe to query, since its associated data and outgoing edges are final. When the scope is closing, queries over labels that are not weakly critical, i.e. that are not in O, are still safe to execute. The idea is that once the scope is closing, the set of open labels only decreases, and only labels in O are weakly critical.

Figure 4 shows the state transition diagram for scope states, and Figure 5 shows the pre- and post-conditions for the transitions. Initially, scopes are in the open state. In this state, the set of open labels has not been initialized, so all labels are considered weakly critical. Initialization with $initScope(d, L)$ changes the state to closing. The flag d specifies if the scope will have associated data. The set of edge labels L determines which labels outgoing edges from this scope may have. In the state closing, the set of open labels, and therefore the set of weakly critical edges, only decreases. The preconditions on *setDatum* and $addEdge(l)$ guarantee that the shape of the scope with respect to a label l only changes when the label is in the set of open labels O. The associated data of a scope can only be set once, therefore *setDatum* always removes the data label \$ from the set of open labels. Edge labels are closed with $closeEdge(l)$, which removes that label from the set of open labels, after which no new edges with that label are allowed, and the label is not weakly critical anymore. After all labels have been closed, the scope is complete and in the state closed.

4 Hierarchical Compilation Units

In this section we introduce a model of hierarchical compilation units. We extend scope states with a notion of sharing that is required by this model. Finally, we present the API of our framework, and code samples for the type checkers that may check our running example.

4.1 The Compilation Unit Model

We propose a model of hierarchical compilation units. The goal of this model is to be flexible enough to handle many different project structures. Examples of typical project structures that we support are:

- a flat set of compilation units for the source files in the project, each of which introduces global declarations that are accessible from other source files,
- a tree of compilation units, where intermediate nodes represent packages or modules, and the individual source files are the leaves, or
- a project which depends on a library that is otherwise independent of the project.

Each compilation unit in our model has an associated, user defined, type checker. Compilation units can spawn sub-units, with their own associated type checkers. Each compilation unit builds a local scope graph by creating scopes, setting data, and adding edges. The compilation units are connected via scopes that are shared between units and their sub-units. Shared scopes allow sub-units to provide declarations that are reachable for other units, and to resolve to names in other units. The examples above all fit into this model. A project with a flat structure consists of one project unit, which creates a global scope that is shared with all file units, which add globally reachable declarations to the global scope. A project with hierarchical packages has compilation units for each package level, each with their own package scope, which is declared in the scope of the parent package. In a project consisting of a library and a program that depends on it, the program and the library have their own root scopes, and the dependency is reflected by an edge from the program root scope to the library root scope.

The compilation units for our Java example of Figure 2 follow the package hierarchy. Figure 6 shows how the scope graph of our example is distributed over compilation units. Our example program has five compilation units, which are depicted by the dashed boxes. At the top level, surrounding the whole scope graph, is the unit that represents the whole program. The packages `p` and `q` are sub units, each containing the units for the class in that package. The *owner* of a scope or edge in the graph is the unit that created the scope or edge. For example, the root scope s_R is owned by the root unit, while the class scope s_B is owned by the innermost unit for the class `B`. Visually, a scope is owned by the innermost unit that contains it, while an edge is owned by the innermost unit that contains the edge label. The MTHD edge is therefore owned by the unit of `B`, as is the LEX edge to s_R.

4.2 Safe Name Resolution with Sharing

The connection between units and sub-units is established through scopes that are shared from a unit to its sub-units. As a result, multiple units may contribute to a scope, something that the unit owning the scope must take into account when handling queries in that scope. Therefore, we extend scope state with sharing, to account for the fact that scope state is determined by the owner as well as by sub-units with which the scope was shared.

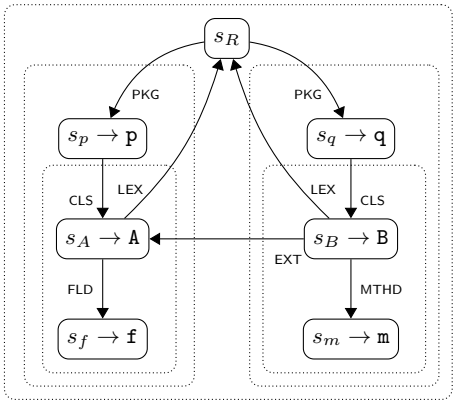


Figure 6 Compilation units for the Java program of Figure 2. The dashed boxes indicate the boundaries of the compilation units. A scope is owned by the innermost unit in which it appears. Edges are owned by the innermost unit that contains their label.

Sharing can result in outgoing edges having a different owner than their source scope, as units can contribute outgoing edges to either their own scopes, or scopes owned by one of their enclosing units that are *shared* with the unit. In our example, the CLS edge to s_B has source scope s_q, which is owned by the unit of package q, not by the unit of B. The data associated with scopes can only be provided by the owner.

The compilation unit that owns a scope is responsible for executing queries on that scope. Every unit maintains an aggregate view of each scope it owns, consisting of all the edges contributed by itself or by sub-units that the scope is shared with. To ensure safe name resolution in this model, we must account for sharing of scopes between units. When a unit initializes one of its scopes, it does not necessarily know what edges any of the sub-units may contribute. The sub-units must therefore initialize shared scopes as well, so that the scope owner has a complete picture of the state of the scope. In the previous section, a scope moved immediately to the closing state when it was initialized. When the scope is, or can be, shared, this is not correct. When a scope is not in state open, we expect that the set of open labels only decreases. The initialization of the scope by a sub-unit could increase the set of open labels. On top of that, if the scope is shared with a new sub-unit, this sub-unit must now also initialize the scope, potentially adding open labels. We can be sure that the set of open labels will only decrease, when all units that the scope is shared with have initialized it, and none of those units will share the scope with new sub-units.

To handle sharing correctly, we extend scope states with an explicit notion of sharing. The state diagram for scope states with sharing is shown in Figure 7, and the pre- and postconditions for the transitions in Figure 8. The extended scope state consists of a set of open labels per unit O, a set I of units that must initialize the scope, and a set H of units that may share the scope with new sub-units. All transitions take a parameter u that indicates which unit is responsible for the event. Creation of a scope is indicated by $new(u, d)$, where u is the owner and the flag d indicates whether the scope has associated data. The flag d is not part of *initScope* anymore, because only the owner can set data, but the scope is initialized by all units that the scope is shared with. When a scope is shared with a unit $\hat{u}$ with $shareScope(u, \hat{u})$, that unit is added to the set I. Every unit that the scope is shared with must initialize it with $initScope(u, L, h)$, after which it is removed from I. A unit initializes the scope with the set of open labels L, a well as the flag h to indicate that the unit may share the scope with sub-units. The scope moves to the state closing when the set

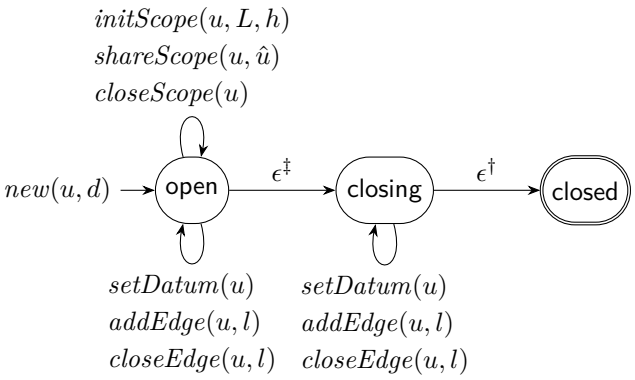


Figure 7 Scope states with sharing. Transition diagram.

$$
\begin{array}{rcl}
\{\top\} & new(u,d) & \left\{\begin{array}{l} I = \{u\}, \\ O = \{\$ \mid d = \top\}, \\ H = \emptyset \end{array}\right\} \\
\left\{\begin{array}{r} u \in I, \\ O = O', \\ u \notin H \end{array}\right\} & initScope(u,L,h) & \left\{\begin{array}{l} u \notin I, \\ O = O' \uplus \{(u,l) \mid l \in L\}, \\ u \in H \end{array}\right\} \\
\left\{\begin{array}{r} u \in H, \\ \hat{u} \notin I \end{array}\right\} & shareScope(u,\hat{u}) & \{\hat{u} \in I\} \\
\{u \in H\} & closeScope(u) & \{u \notin H\} \\
\{(u,\$) \in O\} & setDatum(u) & \{(u,\$) \notin O\} \\
\{(u,l) \in O\} & addEdge(u,l) & \{\top\} \\
\{(u,l) \in O\} & closeEdge(u,l) & \{(u,l) \notin O\} \\
\{I = \emptyset, H = \emptyset\} & \epsilon^{\ddagger} & \{\top\} \\
\{O = \emptyset\} & \epsilon^{\dagger} & \{\top\}
\end{array}
$$

Figure 8 Scope states with sharing. Transition conditions and effects.

of uninitialized units I and the set of sharing units H are both empty. When the state is not open anymore, the set of open labels will only decrease. The events $setDatum(u)$ indicates associated data is set, while $addEdge(u, s, l, s')$, and $closeEdge(u, s, l)$ indicate adding an edge and closing an edge label. The preconditions require that these events are only coming from units that have already initialized the scope. Note that these events are allowed in the states open *and* closing. This ensures that, if a scope is shared between multiple units, each unit can extend that scope without having to wait for all other units to initialize the scope first.

4.3 Name Resolution API

The key to support parallel execution of type checkers is to correctly handle the dependencies between compilation units, which result from name resolution. Queries into a unit that has not constructed the relevant part of its scope graph must be delayed, and executed

Algorithm 1 Type Checker and Name Resolution API.

```
1  interface TypeChecker
2  |  function run(S)
3  end
4  interface CompilationUnit
5  |  function freshScope(d) : s
6  |  function addSubUnit(tc, S)
7  |  function initScope(s, L, h)
8  |  function closeScope(s)
9  |  function setDatum(s, d)
10 |  function addEdge(s, l, s')
11 |  function closeEdge(s, l)
12 |  async function query(s, re, D, L) : A
13 end
```

whenever the scope graph is complete enough. Our framework hides this scheduling from type checkers, and thus provides implicit parallel execution. Type checkers are programmed against a name resolution API, shown in Algorithm 1, which contains methods to specify name binding by building a unit's scope graph, resolve names by querying the scope graph, and start sub-units.

Names are resolved with the $query(s, re, \mathbf{D}, \mathbf{L})$ function, which is defined as *async* to reflect the fact that queries cannot always be answered directly by other compilation units. It is up to the type checker to decide if the result should be immediately awaited, or if other work can be done until the answer is available. The framework ensures correct query answers by keeping track of scope states and scheduling queries based on these scope states. Type checkers are responsible for providing the framework with the necessary information to maintain the scope state. The type checker must therefore initialize the set of (locally) open labels and announce whether it may share the scope with sub-units, and it must close edge labels once all edges with that label are added. All the interaction with other units, such as forwarding queries to the right unit, delaying queries, and maintain scope state on sharing is completely hidden from the type checker. For example, the function $addSubUnit(tc, S)$, which starts a sub-unit with the given type checker tc and initial scopes S, takes care of recording the sharing of scopes, and starts the type checker to run in parallel. Type checkers specify what they do locally, the framework implicitly takes care of their parallel execution.

The pseudo code in Algorithm 2 shows how the API could be used to implement a type checker that checks the Java running example.[5] Each type checker is an actor that extends the *CompilationUnit* actor that provides the API, which is explained in detail in Section 5. At the top level is *JavaRootTC*, which takes no scope arguments, and creates the root scope of the project. Initialization specifies no open edge labels, but does allow sharing. For each package a new sub unit is started with a package type checker that takes the root scope as argument. The first is the root scope, which is passed down to the class scopes. After creating the sub units, the scope is closed, to indicate it will not be shared anymore. The package type checkers start by initializing the shared root scope, indicating the scope may

[5] We show all API calls directly, to show how the API can be used. We imagine that in an actual type checker implementation, common patterns of usage would be abstracted away for nicer code.

Algorithm 2 Sketch of a simplified type checker implementation for Java packages and classes. The type checker is defined as compilation units *JavaRootTC* for the project root, *JavaPkgTC* for packages, and *JavaClassTC* for classes. The presented code shows the construction and querying of package and class definitions.

```
1  actor JavaRootTC(P) extends TypeChecker
2      function Run({})
3          s_R := freshScope(⊥)
4          initScope(s_R, ∅, ⊤)
5          for each (p, C) ∈ P do
6              addSubUnit(JavaPkgTC(p, C), {s_R})
7          closeScope(s_R)
8      end
9  end
10 actor JavaPkgTC(⟦package x;⟧, C) extends TypeChecker
11     function Run({s_R})
12         initScope(s_R, {PKG}, ⊤)
13         s_p := freshScope(⊤)
14         initScope(s_p, ∅, ⊤)
15         setDatum(s_p, x)
16         for each c ∈ C do
17             addSubUnit(JavaClassTC(c), {s_R, s_p})
18         closeScope(s_R)
19         closeScope(s_p)
20         addEdge(s_R, PKG, s_p)
21         closeEdge(s_R, PKG)
22     end
23 end
24 actor JavaClassTC(⟦class x extends y { ... }⟧) extends TypeChecker
25     function Run({s_R, s_p})
26         initScope(s_R, ∅, ⊥)
27         initScope(s_p, {CLS}, ⊥)
28         s_c := freshScope(⊤)
29         initScope(s_c, {LEX, EXT, FLD, MTHD}, ⊥)
30         setDatum(s_c, x)
31         addEdge(s_c, LEX, s_R)
32         closeEdge(s_c, LEX)
33         addEdge(s_p, CLS, s_c)
34         closeEdge(s_p, CLS)
35         {(p, z)} := await query(s_c, LEX*CLS, D_x, ...)
36         s'_c := target(p)
37         addEdge(s_c, EXT, s'_c)
38         closeEdge(s_c, EXT)
39         // ... etc ...
40     end
41 end
```

$$
\begin{array}{rcl}
msg & := & \mathit{Start}(S) \mid \mathit{InitScope}(s, L, h) \mid \mathit{ShareScope}(s) \mid \mathit{CloseScope}(s) \\
& \mid & \mathit{AddEdge}(s, l, s') \mid \mathit{CloseLabel}(s, l) \mid \mathit{Query}(s, re, \mathbf{D}, \mathbf{L}) \\
& \mid & \mathit{DeadlockQuery}(u, m) \mid \mathit{DeadlockReply}(u, m, U) \mid \mathit{Deadlocked}(U) \\
token & := & \mathsf{initScope}(s) \mid \mathsf{closeScope}(s) \mid \mathsf{closeLabel}(s, l) \mid \mathsf{answer}(f)
\end{array}
$$

Figure 9 Compilation Unit. Messages and wait-for tokens.

be shared with sub units, and marking PKG as open to allow adding the package declaration. A new package scope s_p is created, with the package name as associated data. The root scope and package scope are shared with the sub units for the classes in the package, after which both scopes are closed. Finally, the package declaration is added to the root scope and the PKG label is closed.

Although not immediately evident in this small example, the fact that the API is fine-grained (e.g., separating closing a scope for sharing from closing an open edge label) allows greater flexibility in how the type checker is implemented than when a type checker would be responsible for aggregating all these events until one final event can be constructed.

The pseudo code for *JavaClassTC* shows a pattern in which scope graph construction and querying are interleaved. The query for the super class is executed and the type checker waits for the result to be able to construct the EXT edge between the class scopes. It is important to realize that, because the framework ensures safe name resolution, this also introduces the possibility of *deadlock*. If, for example, the *JavaClassTC* type checker would postpone $\mathit{closeEdge}(s_c, \mathsf{LEX})$ until after awaiting the query result, the query would get stuck on the still open edge label. It is therefore important to realize that type checker developers are still responsible for scheduling concerns that are part of any compiler implementation (concurrent or not), such as ensuring declarations are introduced before they are queried. The framework cannot solve these issues, as they are dependent on the specifics of the object language, but it ensures the local behavior is preserved when run in parallel. The Statix meta-language [20] provides implicit maintenance of scope state and flexible scheduling as part of the meta-language semantics, so that these concerns can be left implicit in a Statix type system specification. The case study in Section 6 shows that it is possible to implement a Statix solver on top of our framework, which gives the best of both worlds: implicit parallelism *and* implicit handling of scope state and scheduling.

5 Parallel Actor-based Algorithm

In this section we present an algorithm that implements the compilation unit model and API that were introduced in the previous section. First we introduce the actor model that forms the basis of our algorithm, then we discuss the three main aspects of the algorithm:

- maintaining the scope graph and scope states for owned and shared scopes,
- safely resolve queries on own scopes and delegate queries on other scopes, and
- detect deadlock between compilation units to ensure termination.

5.1 Compilation Unit Actors

The algorithm is written following the actor paradigm [1]. Actors are a concurrency model based on message passing. An actor has only local state, and communicates with other actors through messages. Actors are not internally concurrent, and they do not share state. This

Algorithm 3 Compilation Unit. Local actor state.

```
1 actor CompilationUnit()
2     var: scope graph 𝒢
3     var: counting wait-for graph WFG
4     var: delays 𝒵 := ∅
5     abstract function run(S)
```

makes reasoning about concurrency easier with actors than with approaches based on shared state and explicit synchronization.

A compilation unit corresponds to a *CompilationUnit* actor, which definition and local state is shown in Algorithm 3. The members of a *CompilationUnit*, which are introduced in the following sections, are shown in Algorithms 4–6 and 8. The local state of compilation units consists of a scope graph $\mathcal{G}$, a counting wait-for graph WFG, and a set of delayed queries $\mathcal{Z}$. The messages that form the protocol between compilation units are listed in Figure 9. Type checkers are implemented by extending the actor and implementing the abstract *run* method.

Since there are many variations of the actor model, we give a quick overview of the features that we assume in the model:

- Actors are started using **start**, and form a hierarchy. The keywords **self** and **parent** refer to the current actor or its parent actor, respectively. Actor references can be sent to other actors.
- Actors implement **receive** members for all messages that they accept. Inside a message handler, the **sender** keyword refers to the sender of the current message. Messages are sent using **send** *actor*, *msg*. Messages that require a response are sent with **request** *actor*, *msg* and the response is sent from the message handler with **reply** *msg*. Messages from one actor to another are delivered in order, but delivery of messages from different actors is arbitrarily interleaved.
- Actors may implement auxiliary **function** members, which can only be invoked locally.

Some algorithms are presented in an asynchronous style, using futures. They use the following primitives:

- A **future** f represents a value that may be provided later. The value of a future is set by applying it, written as $f(v)$.
- Functions can be marked as **async** to indicate that they return a future. Inside asynchronous functions, the **await** keyword is used to await the results of futures.
- Awaited futures do not block the actor, but suspend the currently handled message and allow other messages to be processed by the same actor. A resumed computation (as a result of a reply or an applied future) always runs in the context of the actor that started it, and never concurrently with message handling or other resumed computations.

The message handlers and functions of the type checker API are implemented in a straightforward way. The handler for the message $AddEdge(s, l, s')$ calls $addEdge(\textbf{sender}, s, l, s')$, and the API function $addEdge(s, l, s')$ calls $addEdge(\textbf{self}, s, l, s')$.

5.2 Maintaining Scope Graph and Scope States

A compilation unit locally maintains its scope graph $\mathcal{G}$ and the states of the scopes it owns. This is done by the group of functions shown in Algorithm 4. These functions are called to handle API calls from the local type checker, or to handle messages received from other units. In the former case, the argument u equals **self**, in the latter u equals **sender**.

Algorithm 4 Compilation Unit. Scope graph.

```
1  function start(S)
2      𝒢 := 𝒢 ⊎ ⟨S, ∅, ∅⟩
3      foreach s ∈ S do waitFor(self, initScope(s))
4      run(S)
5  end
6  function freshScope(u, d)
7      pick s fresh in scopes(𝒢)
8      𝒢 := 𝒢 ⊎ ⟨{s}, ∅, ∅⟩
9      waitFor(u, initScope(s))
10     if d = ⊤ then waitFor(u, closeLabel(s, $))
11     return s
12 end
13 function initScope(u, s, L, h)
14     granted(u, initScope(s))
15     foreach l ∈ L do waitFor(u, closeLabel(s, l))
16     foreach i ∈ 0...h do waitFor(u, closeScope(s))
17     if owner(s) = self then tryReleaseScopeDelays(s)
18     else send parent, InitScope(s, L, h)
19 end
20 function addSubUnit(u, û, S)
21     foreach s ∈ S do shareScope(û, s)
22     start û
23     send û, Start(S)
24 end
25 function shareScope(u, s)
26     waitFor(u, initScope(s))
27     if owner(s) ≠ self then send parent, ShareScope(s)
28 end
29 function closeScope(u, s)
30     granted(u, closeScope(s))
31     if owner(s) = self then tryReleaseScopeDelays(s)
32     else send parent, CloseScope(s)
33 end
34 function setDatum(u, s, d)
35     𝒢 := 𝒢 ⊎ ⟨∅, ∅, {(s, d)}⟩
36     closeLabel(u, s, $)
37 end
38 function addEdge(u, s, l, s′)
39     𝒢 := 𝒢 ⊎ ⟨∅, (s, l, s′), ∅⟩
40     if owner(s) ≠ self then send parent, AddEdge(s, l, s′)
41 end
42 function closeLabel(u, s, l)
43     granted(u, closeLabel(s, l))
44     if owner(s) = self then tryReleaseLabelDelays(s, l)
45     else if l ≠ $ then send parent, CloseLabel(s, l)
46 end
```

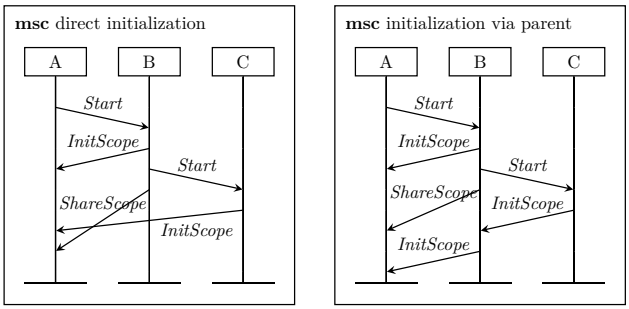


Figure 10 Different initialization scenarios.

Scope state is maintained in a wait-for graph *WFG*, which consists of edges between units, labeled by a token indicating an expected action from the target unit. The tokens that may appear in the wait-for graph are listed in Figure 9. The state of the sets I, H, and O of the scope state is determined by the tokens in the wait-for graph. An $\mathsf{initScope}(s)$ edge to u implies $u \in I_s$. A $\mathsf{closeScope}(s)$ edge to u implies $u \in H_s$. A $\mathsf{closeLabel}(s, l)$ edge to u implies $(u, l) \in O_s$. The state of a scope s can be determined from the tokens in the wait-for graph. If the graph contains $\mathsf{initScope}(s)$ or $\mathsf{closeScope}(s)$ tokens, the scope is open. If the graph only contains $\mathsf{closeLabel}(s, l)$ tokens, the scope is $\mathsf{closing}$. If there are no tokens concerning s, the scope is closed.

The functions in Algorithm 4 update the wait-for graph in correspondence with the postconditions of the scope state transitions. When an element is added to one of the sets of the scope state, an edge is added with $\mathit{waitFor}(u, \mathit{token})$. When an element is removed from one of the sets of the scope state, an edge is removed with $\mathit{granted}(u, \mathit{token})$. For example, when a fresh scope is created with *freshScope*, the function adds an $\mathsf{initScope}(s)$ token, corresponding to the postcondition $u \in I$ of *new*. When the scope is initialized with $\mathit{initScope}(s, L, h)$, the $\mathsf{initScope}(s)$ is removed, corresponding to the postcondition $u \notin I$ of *initScope*.

The removal of tokens from the wait-for graph may result in changes to the weakly critical edges of a scope. Therefore, the functions *initScope*, *closeScope*, and *closeLabel* call one of the *tryRelease*$*$ functions to trigger the release of queries that can now be executed safely.

Maintaining the scope graph and state locally is not enough for a shared scope s that is not owned by the current unit. In such cases, when $\mathsf{owner}(s) \neq \textbf{self}$, the event is propagated to the parent. Because scopes can only be shared with sub units, this means that the message eventually reaches the owner of that scope. The benefit of propagating the message via the parent instead of sending it to the owner of the scope directly has to do with message ordering. Messages coming from two different units are not meaningfully ordered. This can lead to messages arriving in unexpected order, as illustrated by the two scenarios in Figure 10. Without message ordering, a scenario where a top-level unit A, shares a scope with a sub-unit B, which in turn shares that scope with a sub-unit C, could result in A receiving the initialization of C before the message from B that the scope was shared. If the initialization goes via the parent B, then unit A always gets the *ShareScope* message before the corresponding initialization.

Receiving the messages in order simplifies maintenance of the wait-for graph and makes it easier to enforce correct usage of the API in the implementation. This is a simple solution to achieve that without the need for more complex message ordering mechanisms such as vector clocks. Sending messages about shared scopes via the parent is also the reason that

the wait-for graph is a counting graph, that is, tokens may appear multiple times in the graph. To the unit A it looks as if the unit B has to initialize the shared scope twice, as it does not know about the unit C. All messages about sharing and initialization appear to come from B.

5.3 Resolving Queries

The name resolution algorithm, shown in Algorithm 5, implements a graph search that follows well-formed paths to matching declarations. It is a reformulation of the algorithm presented by Van Antwerpen et al. [30]. The entry point is the function $\mathit{query}(p, re, \mathbf{D}, \mathbf{L})$, which returns the environment of paths starting with the prefix path p that matches the given query parameters. The search starts at the target scope of the current path. If the current scope is not owned by the current unit, the query is forwarded to the owner's compilation unit. Otherwise, the environment is computed locally by $\mathit{getEnv}(p, re, \mathbf{D}, \mathbf{L})$. That function determines the set of labels L that is relevant given the current path well-formedness regular expression. Edge labels l are relevant if the Brzozowski derivative [3] does not result in the empty language. If the current regular expression is accepting, that is, its language contains the empty string ϵ, the current scope may be an end-point, and the data label is also relevant. The functions *getEnvForLabels* and *getShadowedEnv* together ensure that the environment implements the label order specified for disambiguation, by ensuring results from more specific labels shadow results from the least specific labels. For example, if the current set of labels $L = \{\$, \mathsf{FLD}, \mathsf{LEX}, \mathsf{EXT}\}$, and the label order is $\$ < \mathsf{FLD} < \mathsf{EXT} < \mathsf{LEX}$, then the resulting environment is

$$\mathit{shadow}(\mathit{shadow}(\mathit{shadow}(A_\$, A_{\mathsf{FLD}}), A_{\mathsf{EXT}}), A_{\mathsf{LEX}})$$

A_l is the answer set for the label l, and *shadow* is the function that removes answers from the right-hand set if its datum matches any answer in the left-hand set. The environment for a single label l is computed by $\mathit{getEnvForLabel}(l, p, re, \mathbf{D}, \mathbf{L})$. If the label is the data label \$, *getDatum* is called in the current scope to construct an answer (p, d). Otherwise, *getEdges* is used to return the target scopes of all outgoing l-labeled edges, cyclic paths are filtered out to ensure search termination, and environments are resolved for each new prefix path p' with the updated path well-formedness. The result is the union of all resulting environments. The updated set of parameters is itself a valid, residual, query, which allows us to simply call the top-level *query* function, which takes care of delegating the query to the right compilation unit.

Compilation units must also ensure that name resolution is safe. When edges or data are requested for which the label is weakly critical, the answer is delayed. When a label's status changes, pending delays are released. The functions *isScopeOpen* and *isEdgeOpen* implement the check for weakly critical edges based on the wait-for graph, as explained in Section 5.2. The functions *getEdges* and *getDatum* decide based on the result of *isEdgeOpen* whether the scope graph can be used, or if it has to delay the answer. If the label is critical, a new future is created, which is stored, together with the scope and label, in the set of delays $\mathcal{Z}$. The functions *tryReleaseScopeDelays* and *tryReleaseLabelDelays* are called whenever the scope state changes, and return the results for any label that is not critical anymore by applying the stored future.

Algorithm 5 Compilation Unit. Query resolution.

```
async function query(p, re, D, L)
    u := owner(target(p))
    if u = self then return await getEnv(p, re, D, L)
    else
        f := request u, Query(p, re, D, L)
        waitFor(u, answer(f))
        A := await f
        granted(u, answer(f))
        return A
end
async function getEnv(p, re, D, L)
    L := {l | 𝓛(∂_l re) ≠ ∅} ∪ {$ | ε ∈ 𝓛(re)}
    return await getEnvForLabels(L, p, re, D, L)
end
async function getEnvForLabels(L, p, re, D, L)
    K⃗ := ∅
    L_max := {l | l ∈ L, ∄l' ∈ L. L(l, l')}
    for each l ∈ L_max do
        L' := {l' | l' ∈ L. L(l', l)}
        K⃗ := K⃗ ∪ {getShadowedEnv(L', l, p, re, D, L)}
    A⃗ := awaitAll K⃗
    return ⋃_{A∈A⃗} A
end
async function getShadowedEnv(L, l, p, re, D, L)
    k_L := getEnvForLabels(L, p, re, D, L)
    k_l := getEnvForLabel(l, p, re, D, L)
    [A_L, A_l] := awaitAll [k_L, k_l]
    return shadow(A_L, A_l)
end
async function getEnvForLabel(l, p, re, D, L)
    if l = $ then
        d := await getDatum(target(p))
        return {a | a = (p, d), D(d)}
    else
        S := await getEdges(target(p), l)
        P⃗ := {p' | s' ∈ S, p' = p · l · s', ∄p'. (p' · l · s prefix of p)}
        K⃗ := {query(p', ∂_l re, D, L) | p' ∈ P}
        A⃗ := awaitAll K⃗
        return ⋃_{A∈A⃗} A
end
function shadow(A_1, A_2)
    return A_1 ∪ {(p_2, d_2) | (p_2, d_2) ∈ A_2, ∄p_1, d_1. ((p_1, d_1) ∈ A_1, d_1 ≈ d_2)}
end
```

Algorithm 6 Compilation Unit. Delays and wait-for graph maintenance.

```
1  function waitFor(u, token)
2  |  WFG := WFG ∪ {(u, token)}
3  end
4  function granted(u, t)
5  |  WFG := WFG − {(u, token)}
6  end
7  function isWaitingFor(u, t)
8  |  return (self, t, u) ∈ WFG
9  end
10 function isScopeOpen(s)
11 |  return ∃u. (isWaitingFor(u, initScope(s)) ∨ isWaitingFor(u, closeScope(s)))
12 end
13 function isEdgeOpen(s, l)
14 |  return isScopeOpen(s) ∨ ∃u. isWaitingFor(u, closeLabel(s, l))
15 end
16 async function getDatum(s)
17 |  if isEdgeOpen(s, $) then
18 |  |  future k
19 |  |  Z := Z ∪ {(s, $, k)}
20 |  |  return await k
21 |  else return data(G)(s)
22 end
23 async function getEdges(s, l)
24 |  if isEdgeOpen(s, l) then
25 |  |  future k
26 |  |  Z := Z ∪ {(s, l, k)}
27 |  |  return await k
28 |  else return {e | e ∈ edges(G), ∃s'.e = (s, l, s')}
29 end
30 function tryReleaseScopeDelays(s)
31 |  if isScopeOpen(s) then return
32 |  foreach {l | ∃k. (s, l, k) ∈ Z} do tryReleaseLabelDelays(s, l)
33 |
34 end
35 function tryReleaseLabelDelays(s, l)
36 |  if isEdgeOpen(s, l) then return
37 |  for each {k | ((s, l), k) ∈ Z} do
38 |  |  Z := Z − {(s, l, k)}
39 |  |  if l = $ then k(data(G)(s))
40 |  |  else k({e | e ∈ edges(G), ∃s'. e = (s, l, s')})
41 end
```

Algorithm 7 Java Type Checker with Incorrect Internal Scheduling.

```
1  actor JavaClassTC(⟦class x extends y { ... }⟧) extends CompilationUnit
2      function Run({s_R, s_p})
3          // ...
4          s_c := freshScope(⊤)
5          addEdge(s_p, CLS, s_c)
6          A := await query(s_c, LEX*CLS, D_x, ...)
7          closeEdge(s_p, CLS)
8          // ...
9      end
10 end
```

5.4 Handling Deadlock

The type checkers implemented with our framework can deadlock for various reasons. The type checker may contain obvious bugs, such as querying a scope before it is properly closed. But many subtle situations can occur as well, if ill-bound or ill-typed input programs cause scope graph construction to get stuck, even if no deadlocks can occur on well-typed inputs.

Whatever the reason, it is important for the user experience to ensure termination of the type checking process and the possibility of graceful handling of deadlocks by the type checker. Our goal is a fine-grained approach where deadlock is handled by failing individual queries that contribute to the deadlock, and only fail whole units as a last resort. Being fine-grained is especially important in interactive settings, when a type checker is employed as part of an IDE. Failing the type checker without returning a result because of an ill-typed input program completely negates the usefulness of the type checker to the programmer in helping them fix their program.

A deadlock occurs when a group of units waits on each other without any unit being able to make progress without receiving a message from one of the other units. We illustrate this using a faulty version of our Java type checker example, shown in Algorithm 7. In this implementation, the super class is resolved before closing the CLS label after the class declaration is added. The program causing deadlock, shown in Figure 11, consists of a class `A` and a class `B` that extends `A`, both defined in a package `p`. The two class definitions are checked by their own units A and B, who declare the classes in the scope s_p that is shared with them by the package unit p. Unit B tries to resolve the class `A` before closing the CLS edge on the shared scope s_p, and the query gets delayed on that edge by unit p. Now the units are in deadlock, since p is waiting for B to close the edge, while B is waiting for an answer from p. We can visualize the dependencies between the units by combining the wait-for graphs *WFG* of all units, as shown in Figure 12. We see that deadlock in the graph from the knot[6] of units that cannot make progress.

[6] In a directed graph, a knot is a set of nodes in the graph such that each node can reach all other nodes in the set. Communication deadlocks are characterized by knots, while resource deadlocks are characterized by cycles.

```
package p;
class A {}
```

```
package p;
class B extends A {}
```

Figure 11 Example program that deadlocks with the buggy type checker from Algorithm 7.

Algorithm 8 Compilation Unit. Deadlock handling.

```
1  function deadlocked(U)
2      if |U| = 1 then
3          if failDelays(U) = false then failAll()
4      else
5          failDelays(U)
6  end
7  function failDelays(U)
8      Z := {f | (u, answer(f)) ∈ WFG, u ∈ U}
9      foreach Z do f(⊥)
10     return Z ≠ ∅
11 end
12 function failAll()
13     for each {t | (u, t) ∈ WFG} do
14         granted(self, t)
15         switch t do
16             case initScope(s) do
17                 if owner(s) ≠ self then send parent, InitScope(s, ∅, false)
18                 tryReleaseScopeDelays(s)
19             case closeScope(s) do
20                 if owner(s) ≠ self then send parent, CloseScope(s)
21                 tryReleaseScopeDelays(s)
22             case closeLabel(s, l) do
23                 if owner(s) ≠ self then send parent, CloseLabel(s, l)
24                 tryReleaseLabelDelays(s, l)
25         end
26 end
```

To understand how we can handle such deadlocks in a fine-grained way, we must understand the shapes these graphs can have. The key insight is that deadlocks involving more than one unit *always* involve a query. If we do not consider queries, the structure of the wait-for graph is always a tree. Units only wait for initScope, closeScope, and closeLabel on themselves or direct sub-units. It is waiting on answers that breaks the tree structure. Therefore, a knot between different units can only exist if at least one query is involved. Our approach handles deadlocks by failing involved queries whenever possible. These failures become exceptions in the type checker, which can be handled if desired. If a deadlock does not involve any queries, and thus involves only a single unit, the whole unit is failed and any remaining open scopes and labels are closed.

The functions for deadlock handling are shown in Algorithm 8. Deadlock detection is implemented using the distributed communication deadlock detection algorithm of Chandy et al. [4], modified so that it collects all units involved in a deadlock. When a deadlock is detected, the *deadlocked* function is called on all units involved, receiving the set U of involved units an

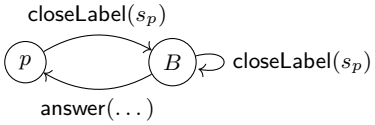
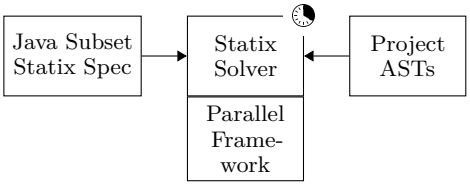


Figure 12 Wait-for graph for the deadlocked example in Figure 11.



Project	#Files	LOC
commons-csv 1.7	12	1845
commons-io 2.6	118	9984
commons-lang3 3.11	210	29642
single-unit-clusters-call	100	~32K

Figure 13 Benchmark setup.

argument. In the case that the set U is a singleton, and the deadlock is local, failing queries is attempted by *failDelays*, and, if unsuccessful, the unit is failed with *failAll*. The function *failDelays* finds all unanswered queries to units in the deadlock and raises an exception locally (indicated by the application of the future with $\perp$). The function *failAll* closes any remaining open scopes and labels and informs the parent if appropriate. At this point the type checker of the failed unit is never invoked anymore, but the unit itself can still resolve queries for other units and participate in deadlock detection. In the case that the set U is not a singleton, the *failDelays* function is used to fail any queries on other units in the deadlock. We explicitly prevent falling back to *failAll* in non-singleton deadlocks because not every unit has queries it can fail. Failing such a unit in such cases would be unnecessary, as some other units in the deadlock can fail queries and resume type checking.

6 Evaluation

We evaluated our approach by porting an existing scope graph-based type checker for a subset of Java to our framework, and measuring speedup resulting from using multiple cores when analyzing Java projects. The diagram in Figure 13 summarizes the setup of the benchmark. The benchmark executable, source code, and data of our experiments can be found in the artifact that accompanies this paper.

6.1 Benchmark

We implemented the type checker by porting the solver of the Statix meta-language to our framework. Adapting the Statix solver was an attractive case study, because it is a mature project that already uses scope graphs for name resolution. The Statix solver uses dynamic scheduling for constraint solving, where constraints are delayed on logical variable instantiation, and was thus a good test case to show that the API provided by the framework is flexible enough to cope with such dynamic scheduling. Therefore, we expect that type checkers in many different scheduling styles can be implemented with our framework, which we plan to explore in future work.

The type checker used a Statix specification for a subset of Java based on an existing MiniStatix specification [20]. This specification focuses on name binding aspects of Java, and implements packages, top-level and nested type definitions, type inheritance. Overloading is partly supported, while generics and lambda expressions are not supported.

We used three existing Java projects (`commons-{csv,io,lang3}`) and one generated Java project for the benchmark (`single-unit-clusters-call`). The existing Java projects are projects from the Apache Commons project that have no dependencies besides the Java standard library (JRE). The projects have different sizes, which allows us to asses the impact of project size on potential speedup. The generated project serves as a baseline for what is achievable with our parallel Statix implementation. It consists of a 100 classes, each class in

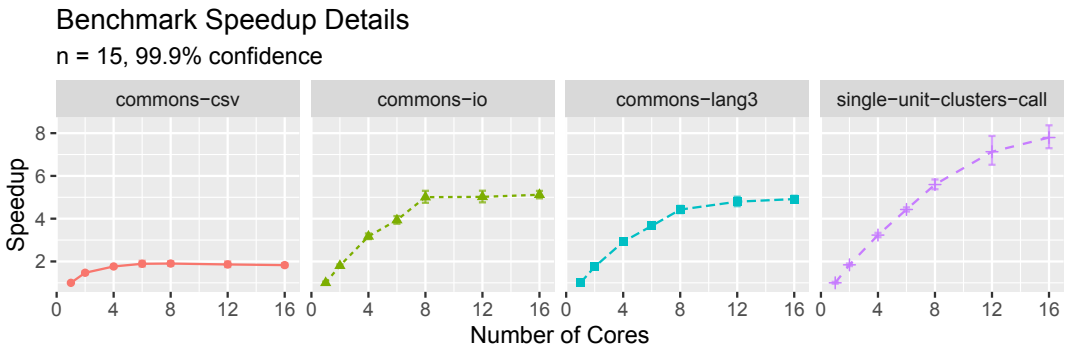


Figure 14 Benchmark results for type checking Java projects. Each subplot shows the speedup, relative to single-core speed, versus the number of used cores for each project. The benchmark was executed with 15 sample iterations, and the error bars represent a 99.9% confidence interval.

its own package. The classes contain a number member methods, and each method body consists of method calls to other members of the same class. Because the classes are isolated and do not depend on other classes, the resulting compilation units only interact with the package's compilation unit and the unit for the Java standard library, and represent an ideal scenario in terms of parallelization. All projects and project sizes are listed in Figure 13.

Early experiments showed that the JRE, which is also treated as a compilation unit, often became the critical unit if it was served by a single actor. To eliminate this effect, the JRE is hosted on as many actors as the number of used cores, using round-robin scheduling to distribute queries over the actors. This is possible because the scope graph for the JRE is precomputed and statically loaded at the start of type checking.

We ran our type checker on each of these projects using an increasing number of cores. The benchmarks were executed using the JMH benchmark tool [17] in single-shot mode (the analysis was run once per iteration) using 5 warm-up and 15 measurement iterations. The benchmarks were executed on a Linux system with 128 AMD EPYC 7502 32-Core Processors 1.5GHz and 256GB RAM.

The results, shown in Figure 14, show the speedup of the parallel type checker, relative to the single core case, for the number of used cores. The error bars indicate the 99.9% confidence interval.

First, we see that the generated baseline project scales up to 5.6x for 8 cores. The scaling slows down more cores are used, but keeps increasing to ~7.8x for 16 cores.

Second, we see that the other projects all have a cut-off point after which adding more cores does not result in much speedup. The cut-offs are approximately at 4 cores for `commons-csv`, the smallest of the three, with a speedup of 1.8x, at 8 cores for `commons-io`, with a speedup of 5.0x, and at 8 cores for `commons-lang3`, with a speedup of 4.42x.

The cut-off in scaling can be explained by looking at the run time of individual compilation units. All projects contain a few source files that are significantly larger than most others. The cut-off happens when the run time of the whole problem is dominated by the run time of the largest compilation unit. If we look at the speedups discussed before, the run time of the longest-running unit as a percentage of the total run time was 84% for `commons-csv`, 100% for `commons-io`, and 81% for `commons-lang3`. Understanding why scaling slows down for some projects before the longest-running unit *completely* dominates the run time is an interesting question for future research.

These results suggest that our approach can give significant speedups for the Statix type checker. How well the approach scales depends on the type checker implementation as well as the granularity of parallelism. Our choice to parallelize on files means that the distribution of file sizes is important for the speedup that can be achieved. A type checker that supports more fine-grained parallelization (e.g., on method bodies), could possibly scale further. Our framework does not require file granularity and supports more fine-grained parallelism. Thus, users can experiment with the granularity that works well for their target language.

Note that these results are for a single type checker and for a single programming language. Both the implementation of the type checker and the design of the language may influence the possibility for effective parallel execution. The relation between language design, the resulting dependency patterns between compilation units, and the opportunity for parallelization is an interesting topic for future research. Our framework enables such experiments with parallel type checkers, by taking the hard parts of parallelization away from the compiler writer.

6.2 Supporting Local Inference

The Statix solver uses unification, and often relies on unification variables in scope graph data to be able to do inference. This posed a challenge when porting Statix to our framework. Our framework operates under the assumption that compilation units only communicate via the scope graph. This means the unifier of one compilation unit is not accessible to other compilation units. While the owning compilation unit can interpret that data relative to the local unifier, other units can not. We have a situation where a unit requires an incomplete view of its own data, but other units should only ever get the complete data.

We added a small extension to the framework to support such local inference patterns. Type checkers can define a function that produces a representation of data that is fit for other units:

async function *GetExternalRepresentation(d)*

The function is asynchronous so the type checker can delay returning the external representation until unification variables are instantiated. It is applied to the data of any scope whose owner is not the unit that issued the query. This solution allows units to do local type inference via the scope graph, while still presenting complete data to other units.

In the Statix literature, different patterns are used to associate declarations with types. In the first, the declaration and type are combined as a tuple $(x : T)$, and stored as the data in a single scope [31]. In the second, the declaration only carries the name as data, and the type is represented as the data of a separate scope connected to the declaration by an edge [20]. We observed that the first encoding quickly results in deadlock if name resolution queries (resolve x) are necessary to instantiate the types T: The external representation of the whole tuple gets stuck on the logical variables in the type, therefore blocking the query for the name. The representation using tuples can easily be converted into the latter, but is a necessary consequence of the isolated nature of the compilation units in our approach.

7 Related Work

In this section we discuss related work on parallel approaches to build systems, compilers, and program models used for compiler and static analysis implementation.

7.1 Parallel Compilers

Parallel compilers are certainly not a given, even for often used languages, but there are several languages for which parallel compilers (mature or research prototypes) exist. These compilers are all for specific languages, but it is interesting discuss the techniques they use or the performance results they achieve. Although it is hard to find reliable information on the parallel capabilities of compilers, online discussion in StackExchange suggest that compilers for at least Java, C/C++, and C#, all often used, are all single-threaded [24, 25, 26]. The concurrent compiler for Active Oberon [18] implements ideas that are similar to ours. Scopes (following the program nesting structure) have an associated state describing whether all symbols in the scope have been defined, and queries are delayed if scope information is incomplete. The supported scoping structure is specific for the target language and deadlock is avoided by being careful about what queries are done in what compilation phase. The implementation uses a shared data structure for the symbol table with a global lock, which is different from our approach of a distributed scope graph and units communicating by messages only. Hydra [29] is a commercial parallel compiler for Scala, which parallelizes the Scala compiler by running the many phases of the Scala compiler in parallel. Hydra publishes benchmark results and reports speedups between 1.8–3.5x, depending on the project, on 4 cores [29]. Work has been done to parallelize the Rust compiler [21]. The approach is focused in parallelizing loops in the compiler, while maintaining most of the current structure of the compiler. However, at the time of writing the documentation mentions that "work on explicitly parallelizing the compiler has stalled. There is a lot of design and correctness work that needs to be done." The Go compiler supports parallel compilation at certain levels of the program [7]. Particularly, the compilation of functions inside a package is executed in parallel. Finally, the Swift compiler takes an interesting approach to achieving parallel build [28]. Every compilation task has a focus, the compilation unit it "really" needs to compile. In the process it also compiles other units, but only as much as necessary for the focus unit. They claim this generally works well, because the necessary work on other units is limited.

7.2 Parallel Build Systems

Our framework shares many characteristics with build systems, as they run and order compilation tasks based on a dependency graph. The well known build tool Make [27] executes build tasks based on a statically known acyclic dependency graph. When the object language allows separate compilation, it can run these tasks in parallel as well. Many other build tools follow the model of Make, and require the dependencies to be acyclic and known a-priori. Some build tools such as Pluto [5] and PIE [10] improve on this model by supporting dynamic dependency discovery. The resulting dependency graph is still required to be acyclic. What all these have in common is that the build tasks are all treated as atomic operations, producing outputs from inputs. The build system is concerned with ordering these tasks correctly. This is in contrast with our approach, which makes partial results of a unit available to other units before it is completely finished. This allows us to support not only dynamic dependencies, but also cycles in the dependency graph, something that build systems cannot handle.

7.3 Parallel Programming Models

Another approach is to write the compiler in a programming model that supports parallel execution, and the parallelization is not organized around compilation units anymore.

The JastAdd framework for reference attribute grammars supports implicitly parallel attribute evaluation [34]. The resulting concurrency is more fine-grained than in our approach, and not necessarily driven by dependencies between compilation units. If one writes a compiler using reference attribute grammars, this is a convenient way to parallelize the compiler. Compared to our approach, reference attribute grammars do not provide a ready to use model for name binding. This means it falls on the developer to come up with suitable representations and algorithms for the object language. Applying parallel attribute evaluation to the ExtendJ Java compiler resulted in speedups of 1.52–2.43x on 4 cores. Although their evaluation was done on a different set of Java projects, these results suggest that the performance of our approach is competitive.

LVish [13] proposes a parallel programming model based on monotonically growing data and freezing variables that reached a final state to achieve quasi-deterministic parallelism. It has been successfully applied to parallel type inference [15]. This model is very similar to how we handle scope states: closing scopes and edges corresponds to freezing. The difference is that LVish is only a model for monotone state, which leaves users to build parallelization around it. The scope state model is specialized for our purpose, which allows us to make the parallelization and deadlock handling implicit for the user.

The theorem prover Isabelle/PIDE has strong support for implicit parallelization of proof checking [32, 14, 33]. The granularity is much smaller than in our approach. They do not support cyclic dependencies between parallel task, but a high degree of parallelism is achieved by exploiting proof irrelevance: most dependencies are only on the level of the theorem statements, but not their proofs. They report speedups up to 5.2–6.4x on 8 cores [33].

Several parallel programming models have been developed targeting static analyses. Because of their focus, these approaches target certain kinds of computations that are commonly used in static analyses, such as fixed points over lattices [8], parallel iteration over sets [12], or established algorithms such as IDFS [19].

7.4 Scope Graphs

Scope graphs [16] were introduced as a language-independent model of name binding with a focus on expressive, non-lexical, binding patterns, formalizing and generalizing the semantics of NaBL [11]. This model was subsequently extended and used to develop formalisms for the specification of type checkers [30, 31], resulting in the meta-language Statix. Followup work [20] defined a formal, non-parallel, operational semantics for Statix, and proved it correct. It introduced the notion of *critical edges* as a tool to reason about query answer correctness in evolving scope graphs. Critical edges were defined in terms of the constructs of the Statix language and the presented operational semantics. In this paper we introduce *scope state* as an explicit and application independent description of the state and transitions of a scope in a evolving scope graphs, which was only implicitly present in the Statix operational semantics. Porting Statix to the parallel framework of this paper required reformulating the safety conditions of the original operational semantics to explicit scope state operations. All this work has been developed and applied in the context of the Spoofax language workbench [9].

8 Conclusion

In this paper we have introduced a framework for the implementation of implicitly parallel type checkers. We have introduced the concept of scope state to make the notion of weakly critical edges in evolving scope graphs explicit. We have presented a case study and shown that the approach does result in speedups for the larger projects in our benchmark. For

all real-world projects in the benchmark the scaling was limited by a few large files, which suggest that more fine-grained parallelism (e.g., checking method bodies in parallel) could improve parallelism for this Java type checker. In general, investigating the relation between the type checker implementation/Statix specification and the achievable parallelization for different target languages is interesting follow-up research. Other interesting directions for future research are (a) extending this work to *incremental* type checking of large software projects during development, (b) developing useful abstractions for managing scope state that sit between the fine-grained API of this paper, where scope state is completely explicit, and the high-level abstraction offered by the Statix meta-language, where scope state and evaluation order are completely implicit, and (c) investigating how this work can be extended to and/or integrated with other compiler tasks such as parsing, and code generation to create a fully parallelized compiler pipeline.

References

1. Gul A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990.
2. Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.
3. Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.
4. K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1(2):144–156, 1983. doi:10.1145/357360.357365.
5. Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. A sound and optimal incremental build system with dynamic dependencies. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 89–106. ACM, 2015. doi:10.1145/2814270.2814316.
6. GCC. The parallel gcc. URL: https://gcc.gnu.org/wiki/ParallelGcc.
7. Go. Go 1.9: Parallel compilation. URL: https://golang.org/doc/go1.9#parallel-compile.
8. Dominik Helm, Florian Kübler, Jan Thomas Kölzer, Philipp Haller, Michael Eichberg, Guido Salvaneschi, and Mira Mezini. A programming model for semi-implicit parallelization of static analyses. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 428–439, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3395363.3397367.
9. Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM. doi:10.1145/1869459.1869497.
10. Gabriël Konat, Sebastian Erdweg, and Eelco Visser. Scalable incremental building with dynamic task dependencies. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 76–86. ACM, 2018. doi:10.1145/3238147.3238196.
11. Gabriël Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2012. doi:10.1007/978-3-642-36089-3_18.
12. Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In Jeanne Ferrante and Kathryn S.

McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 211–222. ACM, 2007. doi:10.1145/1250734.1250759.

13 Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze after writing: quasi-deterministic parallel programming with lvars. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 257–270. ACM, 2014. doi:10.1145/2535838.2535842.

14 David C. J. Matthews and Makarius Wenzel. Efficient parallel programming in poly/ml and isabelle/ml. In Leaf Petersen and Enrico Pontelli, editors, *Proceedings of the POPL 2010 Workshop on Declarative Aspects of Multicore Programming, DAMP 2010, Madrid, Spain, January 19, 2010*, pages 53–62. ACM, 2010. doi:10.1145/1708046.1708058.

15 Ryan R. Newton, Ömer S. Agacan, Peter P. Fogg, and Sam Tobin-Hochstadt. Parallel type-checking with haskell using saturating lvars and stream generators. In Rafael Asenjo 0001 and Tim Harris, editors, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, page 6. ACM, 2016. doi:10.1145/2851141.2851142.

16 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. doi:10.1007/978-3-662-46669-8_9.

17 OpenJDK. Java Microbenchmark Harness (JMH). URL: https://openjdk.java.net/projects/code-tools/jmh/.

18 Patrik Reali. Structuring a compiler with active objects. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages, Joint Modular Languages Conference, JMLC 2000, Zurich, Switzerland, September 6-8, 2000, Proceedings*, volume 1897 of *Lecture Notes in Computer Science*, pages 250–262. Springer, 2000.

19 Jonathan Rodriguez and Ondrej Lhoták. Actor-based parallel dataflow analysis. In Jens Knoop, editor, *Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6601 of *Lecture Notes in Computer Science*, pages 179–197. Springer, 2011. doi:10.1007/978-3-642-19861-8_11.

20 Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428248.

21 Rust. Parallel compilation. URL: https://rustc-dev-guide.rust-lang.org/parallel-rustc.html.

22 V. Seshadri, David B. Wortman, Michael D. Junkin, S. Weber, C. P. Yu, and I. Small. Semantic analysis in a concurrent compiler. In *PLDI*, pages 233–240, 1988.

23 Zhong Shao and Andrew W. Appel. Smartest recompilation. In *POPL*, pages 439–450, 1993.

24 StackExchange. Do compilers utilize multithreading for faster compile times? URL: https://softwareengineering.stackexchange.com/questions/322494/do-compilers-utilize-multithreading-for-faster-compile-times.

25 StackExchange. Is there something that prevents a multithreaded c# compiler implementation? URL: https://softwareengineering.stackexchange.com/questions/330026/is-there-something-that-prevents-a-multithreaded-c-compiler-implementation.

26 StackExchange. Why isn't javac running on multiple cores? URL: https://stackoverflow.com/questions/46461757/why-isnt-javac-running-on-multiple-cores.

27 Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU Make*. Free Software Foundation, May 2016.

28 Swift. Swift compiler performance. URL: `https://github.com/apple/swift/blob/master/docs/CompilerPerformance.md`.

29 Triplequote. Hydra: The parallel scala compiler. URL: `https://triplequote.com/hydra/compilation/`.

30 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In Martin Erwig and Tiark Rompf, editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 49–60. ACM, 2016. `doi:10.1145/2847538.2847543`.

31 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018. `doi:10.1145/3276484`.

32 Makarius Wenzel. Parallel Proof Checking in Isabelle/Isar. In *ACM SIGSAM Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS '09)*, page 9, New York, NY, USA, 2009. Association for Computing Machinery.

33 Makarius Wenzel. Shared-memory multiprocessing for interactive theorem proving. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 418–434. Springer, 2013. `doi:10.1007/978-3-642-39634-2_30`.

34 Jesper Öqvist and Görel Hedin. Concurrent circular reference attribute grammars. In Benoît Combemale, Marjan Mernik, and Bernhard Rumpe, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, pages 151–162. ACM, 2017. `doi:10.1145/3136014.3136032`.

Lossless, Persisted Summarization of Static Callgraph, Points-To and Data-Flow Analysis

Philipp Dominik Schubert
Heinz Nixdorf Institute, Paderborn, Germany

Ben Hermann
Technische Universität Dortmund, Germany

Eric Bodden
Heinz Nixdorf Institute, Paderborn, Germany
Fraunhofer IEM, Paderborn, Germany

Abstract

Static analysis is used to automatically detect bugs and security breaches, and aids compiler optimization. Whole-program analysis (WPA) can yield high precision, however causes long analysis times and thus does not match common software-development workflows, making it often impractical to use for large, real-world applications.

This paper thus presents the design and implementation of ModAlyzer, a novel static-analysis approach that aims at accelerating whole-program analysis by making the analysis modular and compositional. It shows how to compute *lossless*, persisted summaries for callgraph, points-to and data-flow information, and it reports under which circumstances this function-level compositional analysis outperforms WPA.

We implemented ModAlyzer as an extension to LLVM and PhASAR, and applied it to 12 real-world C and C++ applications. At analysis time, ModAlyzer modularly and losslessly summarizes the analysis effect of the library code those applications share, hence avoiding its repeated re-analysis. The experimental results show that the reuse of these summaries can save, on average, 72% of analysis time over WPA. Moreover, because it is lossless, the module-wise analysis fully retains precision and recall. Surprisingly, as our results show, it sometimes even yields precision superior to WPA. The initial summary generation, on average, takes about 3.67 times as long as WPA.

2012 ACM Subject Classification Software and its engineering → Automated static analysis

Keywords and phrases Inter-procedural static analysis, compositional analysis, LLVM, C/C++

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.2

Funding This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre 901 "On-The-Fly Computing" under the project number 160364472-SFB901/3 and the Heinz Nixdorf Foundation.

1 Introduction

Static analysis plays an important role in modern software development. While intra-procedural static data-flow analysis might only be useful in a limited number of use-cases, inter-procedural analysis is a powerful building block for bug finding [4, 7, 34], compiler optimization [6, 8] and software hardening [22, 39, 40, 44, 47].

Static analysis is known to be an undecidable problem [57], which challenges static-analysis designers to define analyses that are both precise (yielding little to no approximate information) and efficient. To obtain good precision, static program analyses need to be inter-procedural, i.e., cross procedure boundaries, and also must be context sensitive [68]. Moreover, they must be based on precise points-to analyses [26].

Such inter-procedural analysis, however, especially if implemented as a whole-program analysis (WPA), is notorious for causing problems with scalability in both runtime and memory consumption. The memory consumption required for larger programs to keep the complete program representation as well as all of the data structures required to perform the analyses and optimizations in memory can easily grow to a large two-digit GB figure [13, 80]. Analysis times can amount to several hours, impeding development processes even in cases where the analysis is deployed as nightly build [23, 46, 69].

There are application scenarios for which one can yield useful results with *intra-procedural* analyses that are simple enough to scale. The clang-tidy tool [2] and Cppcheck [5] use *syntactic* analyses that are able to analyze software comprising a million lines of code within minutes. Many *semantic* program analyses, however, such as data-flow [42], typestate [74, 75] or shape analyses [82], for instance, require detailed program representations that incorporate the effects of procedure calls, yet are virtually impossible to scale if computed for the whole program at once. This precludes important application scenarios, for instance, IDE integration or the automated scanning of frequently changing software. Facebook, for instance, reports that its code base changes so frequently that it has become a real challenge to design analysis tools such that they can report errors quickly enough so that they are still relevant and actionable when reported [37].

In this work, we aim to scale static context-, flow-, and field-sensitive inter-procedural program analysis using a compositional computation of analysis information. The effectivity of this compositional program analysis depends on the number of reusable parts of an application, e.g., program parts that constitute frameworks or libraries, or for parts that simply do not change from one analysis run to the next. A recent study by Black Duck (Synopsis) has shown that more than 96% percent of the applications they scan contain open-source components and that those components now make up, on average, 57% of the code [70]. As those dependencies are updated much less frequently than application code, compositional analysis can potentially accelerate the analysis of applications by reusing analysis results from previous runs.

Previous work on compositional program analysis has been restricted to certain types of data-flow analysis only. Reviser [20], for instance, allows for the ahead-of-time computation of reusable taint-analysis summaries for Java libraries. Reviser builds on concepts by Rountev et al. [62], who showed how to obtain reusable libraries for general distributive data-flow problems. Both those previous approaches, however, have two significant limitations: First, they only apply to Java, making it unclear which concepts carry over to other languages, particularly C/C++, which allow more liberal pointer accesses to the stack and heap. Second, they only apply to data-flow analysis and leave out the composition of points-to and callgraph information. Especially the latter is a serious practical limitation: when composing a library summary with application code, these approaches again perform an expensive whole-program points-to and callgraph analysis, which in itself can take several minutes if not hours to complete. In result, these approaches incrementalize only the tip of the proverbial iceberg. Addressing this limitation is complex as callgraph, points-to, and data-flow information are inter-dependent. A core conceptual contribution of this paper is therefore also *a mechanism for analysis dependency management for a fully compositional analysis*. This mechanism automatically triggers updates whenever novel information becomes available that affects existing information.

An important practical factor impacting the scalability of compositional analysis is the mechanism to persist summaries. While the approach by Rountev et al. [62] *computes* summaries, they are *not persisted at all* [58] but rather discarded at analysis shutdown, which

completely defeats their purpose. Reviser [20] does persist summaries, but its summary format is only applicable to taint analysis that uses a binary lattice $\frac{\top}{\bot}$. Finding an efficient summary format that is able to persist general data-flow information is challenging due to arbitrarily complex lattices used by more advanced analyses. However, efficient and generalized persistence of summaries is key to effective compositional analysis.

This paper presents ModAlyzer, a novel approach to compositional analysis that in contrast to earlier approaches performs an *integrated compositional analysis* for callgraph, points-to and context-sensitive data-flow information in a module-wise fashion. ModAlyzer allows the compositional pre-computation of all three pieces of information for individual C/C++ modules, such as libraries and frameworks. Information precomputed this way is then efficiently persisted, and later-on *merged* into larger analysis scopes. Merging analysis information efficiently is an integral part of any compositional analysis approach as combining analysis information computed on individual pieces of code is required to produce overall analysis results.

As our experiments show, this frequently helps to achieve a more efficient analysis of entire applications (compared to WPA) while retaining the same level of precision and recall of a matching WPA.

Interestingly, as this paper shows, merge operations on different types of analysis information can be modelled in a common way by defining merge operations on their respective graph representations. ModAlyzer thus conducts its compositional computation of callgraph, points-to, and data-flow information using those graph operations. While ModAlyzer compositionally computes all these kinds of information, it also manages the dependencies among them, and updates dependent information as required. ModAlyzer creates summaries for callgraph and points-to analysis, and for data-flow analyses expressed in the IFDS [55] and IDE [63] frameworks. Those frameworks support data-flow analyses whose flow functions distribute over the meet operator, which in turn allows for an efficient and – as we also show empirically – *lossless* summarization. ModAlyzer does not lose any information and also does not have to overapproximate missing information. Instead, it leaves gaps that will be eventually filled-in during summary application resulting in the same information that would have been obtained by a matching whole program analysis. Many useful data-flow analyses, among others taint analysis as well as all Gen/Kill problems, can be encoded in those *distributive frameworks*. ModAlyzer also allows for the computation of more expressive analyses in the monotone framework [41]. While one generally cannot create *data-flow* summaries for such analyses (an undecidable problem), these analyses nonetheless can benefit from summaries for points-to and callgraph information. This still allows to greatly accelerate analysis computations even for *non-distributive* analysis problems.

We have implemented ModAlyzer on top of PhASAR [64] and LLVM [45]. We show the improvements of ModAlyzer's compositional analysis over traditional whole-program analysis by analyzing 12 real-world C/C++ applications of various sizes, reaching from 129,000 to 1,400,000 lines of code. For each application, we perform two client analyses (uninitialized-variables analysis and taint analysis), once in whole-program mode and once using library summaries pre-computed by ModAlyzer. We compare the resulting running times and client reports to validate the equivalence in precision and recall, and to assess analysis time. Our experiments show that ModAlyzer can decrease the analyses' runtimes between 28% and 91% while keeping the initial one-time runtime overhead for summarization of library parts at 3.67 times as long as the cost of a whole-program analysis.

We will make the implementation of ModAlyzer available as open source under the permissive MIT license. We subject it to artifact evaluation. All accompanying artifacts of this paper, including the processed target applications, their modularizations, and result data are available online under the MIT license [16].

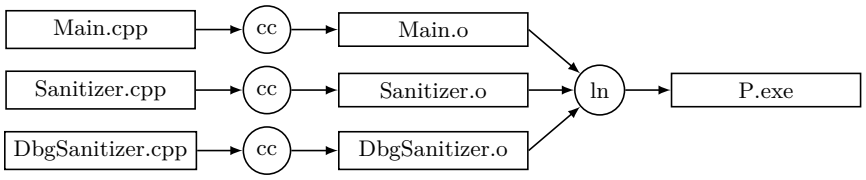


Figure 1 C/C++'s compilation model. cc is the C/C++ compiler. ln is the linker.

In summary, this paper makes the following contributions: it presents

- the first *integrated compositional analysis* for callgraph, points-to and context-sensitive data-flow information with appropriate summarization techniques and summary formats,
- ModAlyzer, an open-source C++ implementation within the PhASAR [64] framework, allowing the full module-wise computation of arbitrary distributive static analysis problems (and module-wise computation of points-to and callgraph information for non-distributive analysis problems),
- and an experimental evaluation of ModAlyzer, which shows that not just in theory but also in practice precision and recall are retained, and which assesses under which circumstances the reuse of summaries can decrease the overall analysis time.

2 Motivating Example and Intuition

C/C++ programs are usually organized in several files that provide some limited form of modularity. An implementation and its corresponding header file are often referred to as a *compilation unit* or *module*. The compiler translates each module separately and thus, has only knowledge about the information contained within the module that is currently compiled. The resulting object file contains executable program code, which may, however, contain unresolved references. The linker resolves these references across two or more object files and may adds links to external libraries. The result after the linkage step is an executable program. Figure 1 depicts the corresponding mechanism.

The vast majority of modern software is not written from scratch, but rather uses libraries, which enable code reuse, faster development and is less error prone [12, 80]. Thus, only a small amount of a program is actual application code and large parts are library code. Once a library has been introduced as a dependency it is rarely changed compared to the application code that uses it.

Our example program is comprised of three compilation units (CUs) – often called *modules* in the C/C++ context – `Main`, `Sanitizer`, and `DbgSanitizer` shown in Listing 1, 2, and 3. We omit the header files for brevity of presentation. The example program is built according to the compilation model presented in Figure 1.

Let us assume that `Sanitizer` and `DbgSanitizer` form a library for sanitization tasks called `libsan`. In C/C++, a library is a collection of one or more object files that have been compiled in form of an archive or shared object file. We further assume that `Main` represents the user application that makes use of the `libsan` library. We use the example program shown in Figure 2 as a running example to detail on our module-wise analysis (MWA) approach.

As a client analysis we use a *taint* analysis which is able to detect potential SQL injections in programs. A taint analysis tracks values that have been tainted by one or more *sources* through the program and reports a leak, if a tainted value reaches a *sink*. The analysis considers all user inputs of a program which potentially contain malicious data as tainted,

```
int main(int argc, char **argv) {
  auto *con = driver->connect(/* connection properties */);
  auto *stmt = con->createStatement();
  string q = "SELECT name FROM students where id=";
  string input = argv[1];
  string sanin = applySanitizer(input);
  auto *res = stmt->executeQuery(q + sanin);
  res->beforeFirst();
  if (!res->rowsCount()) { cout << "no record found\n"; }
  while (res->next()) { cout << res->getString("name") << '\n'; }
  delete stmt; delete res; delete con; return 0;
}
```

Listing (1) Main – Contains the main application code.

```
struct Sanitizer {
  virtual ~Sanitizer() = default;
  virtual string sanitize(string &in) {
    if (isMalicious(in)) { in = /*actual sanitization*/; }
    return in;
  }
  bool isMalicious(string &in) { return /*check if malicious*/; }
};
string applySanitizer(string &in) {
  Sanitizer *s = getGlobalSan();
  string out = s->sanitize(in);
  return out;
}
```

Listing (2) Sanitizer – A module of the sanitization library.

```
struct DbgSanitizer : Sanitizer {
  bool disable = true;
  ~DbgSanitizer() override = default;
  string sanitize(string &in) override {
    if (!disable && isMalicious(in)) { throw malicious_input(":'("); }
    return in;
  }
};
Sanitizer *getGlobalSan() {
  static Sanitizer *s = new DbgSanitizer;
  return s;
}
```

Listing (3) DbgSanitizer – A module of the sanitization library.

Figure 2 Modular example program.

e.g. the parameters argc and argv that are passed into the main() function in our example program presented in Listing 1. The function Statement::executeQuery() serves as a sink in this scenario. Without sanitization, a malicious user of the program could carefully craft the string "1 OR TRUE;" and pass it as the program's second command-line argument. As the input string is just concatenated the database server will return the names of all students not just the one where the id matches. By crafting such malicious inputs, a user can leak or alter the data stored in the database. A tainted value may be *sanitized* in our scenario by using the Sanitizer :: sanitize () function (Listing 2) that clears malicious contents, and therefore *un-taints* a value. The client analysis $\mathbb{T}$ aims to find flows of (unsanitized) tainted values to sinks and reports a potential SQL injection vulnerability whenever it finds such an illegal flow.

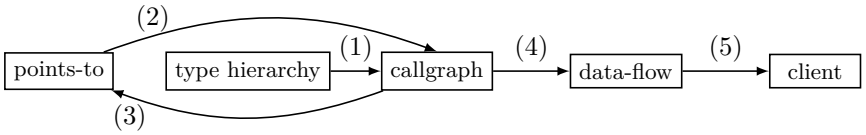


Figure 3 Dependencies of a client analysis involving type hierarchy, points-to information, inter-procedural control-flow and data-flow information. Numbered edges determine computation order.

3 Framework Architecture

In this section, we elaborate on our compositional, *module-wise analysis*. We first present the idea of the algorithm in a nutshell and continue with our concept of summary generation. We then explain the steps we take for result merging and optimizations. As summaries are always depending on assumptions made, we discuss them at the end of this section.

3.1 Idea of the Algorithm

We have built our module-wise analysis approach following C/C++'s compilation model. To determine a program property of interest, a concrete data-flow analysis, the *client*, may require information from other analyses as shown in the dependency graph in Figure 3. To be able to determine the inter-procedural data flow that a concrete client analysis is interested in, a precise callgraph is needed. A precise callgraph, in turn, requires points-to information [26] and the type hierarchy of the program, but points-to analysis requires a callgraph as well. The data-flow information depends on the callgraph and the client analysis transitively depends on all of these pieces of information. Note that a points-to analysis does require information on subtyping. Information such as the declared and allocated pointer types can be queried ad-hoc. Many useful static analyses can be encoded using the dependencies show in Figure 3 and thus, we will assume such a scenario in this paper.

To achieve fully compositional analysis information for all levels of information as shown in Figure 3, we must be able to (i) compute all information required for a client analysis on a function level (except the type hierarchy, which is always computed on a module level) and *summarize* them, (ii) *merge* the information and (iii) perform an *update* if a merge reveals new information that affect the current results. The merge operation combines static analysis summaries computed on two individual modules into a novel summary such that it reflects the information that would have been obtained by linking those modules first and then computing the static analysis information afterwards. In such an MWA-style analysis library modules would be analyzed separately. Their computed summaries would be merged whenever necessary while analyzing a program which uses those library modules.

As mentioned in Section 1, the compositional approaches to static data-flow analysis presented by Rountev et al. [62] and Reviser [20] only apply to Java. In that regard, ModAlyzer can take advantage of C's and C++'s language characteristics, which are quite different from Java. The ModAlyzer approach merges summaries for each function per compilation unit. The intuition is that related source code often resides within the same compilation unit. Because C and C++ are often used to implement performance-critical applications [1, 11], developers have a great interest in making as much information available to the compiler as possible within an individual compilation unit. Otherwise, the compiler would not be able to perform inlining and other important optimizations in an ordinary (i.e., non-WPA) compilation setup [49, 50]. Additionally, whereas all function members (or methods) in Java are virtual, function members are non-virtual by default in C++. It

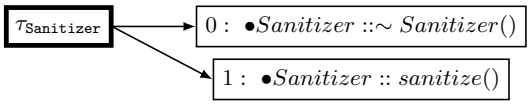


Figure 4 Type hierarchy and respective virtual function table(s) of the `Sanitizer` module.

generally seems that C++ programs make use of dynamic dispatch less frequently to avoid performance penalties [17,29,32], a property that ModAlyzer, again, uses to our advantage. Summaries computed for C/C++ code are thus more expressive and less likely to contain *gaps* due to missing information. While ModAlyzer, in general, is applicable to other languages as well it might work better for C and C++ programs than for programs written in Java or C#, for instance, which use virtual calls all over the place. For those languages, the portion of partial summaries will increase and the overall performance of ModAlyzer will degrade as more gaps need to be closed while analyzing the "main application". Previous works by Rountev show that summarization techniques nonetheless can greatly improve running times for large Java applications, even when restricted to data-flow analysis only. We elaborate on that in detail in Section 5.

In the following, we show that merge operations on analysis information can be modelled in a common way through merge operations performed on their respective graph representations. However, special care must be taken to update the dependent information accordingly if new information becomes available due to merging two module summaries. This makes it crucial to keep particularly the callgraph up to date, as all other information except the type hierarchy depend on it.

3.2 Summary Generation

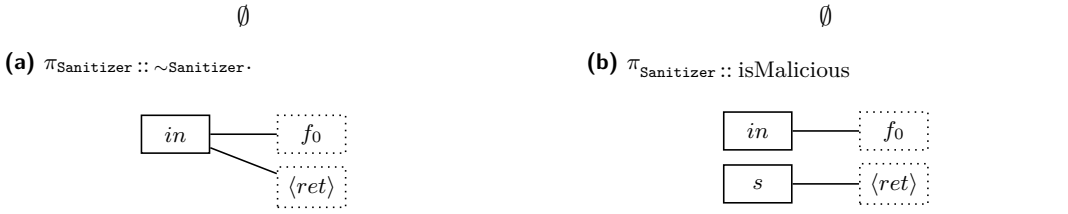
In the following, we will explain the steps of our analysis based on the example presented in Section 2. The assumption is that `Main` changes frequently, and `libsan` only once in a while. For presentation here we start our library pre-analysis by analyzing the `Sanitizer` module, although the analysis algorithm does not make any assumptions about module order.

3.2.1 Type Hierarchies

Our approach first computes the type hierarchy as it is the most robust structure in the sense that the amount of information can only grow monotonically. We use $\tau_{\texttt{t}}$ to denote the type of a class or struct t and we use $T_{\texttt{C}}$ to denote the type hierarchy for a module `C`. In addition, the type hierarchy maintains information on the virtual function tables (call targets) for C++'s *struct* or *class* types that declare virtual functions.

▶ **Example 1.** The analysis will find that the type hierarchy for the `Sanitizer` module consists of a graph containing a single node representing the type $\tau_{\texttt{Sanitizer}}$. The call target for $\tau_{\texttt{Sanitizer}}$ contains two entries, $\{Sanitizer ::\sim Sanitizer(), Sanitizer :: sanitize()\}$.[1] The (partial) type hierarchy for the `Sanitizer` module is shown in Figure 4

[1] If a C++ type is meant to be used polymorphically, its destructor has to be declared virtual. Otherwise, if the static type of an object to be deleted differs from its dynamic type, the behavior is undefined.



(a) $\pi_{\texttt{Sanitizer}} :: {\sim}\texttt{Sanitizer}\cdot$

(b) $\pi_{\texttt{Sanitizer}} ::$ isMalicious

(c) $\pi_{\texttt{Sanitizer}} ::$ sanitize ; f_0 denotes the first formal parameter and $\langle ret \rangle$ the return value of the isMalicious() function.

(d) $\pi_{\texttt{Sanitizer}} ::$ applySanitizer ; f_0 denotes the first formal parameter and $\langle ret \rangle$ the return value of the getGlobalSan() function.

Figure 5 $\Pi_{\texttt{Sanitizer}}$ containing all pointer-assignment graphs of `Sanitizer`.

3.2.2 Intra-Procedural Points-To Information

In the next step, the analysis computes function-wise, intra-procedural, never-invalidating[2] points-to information using an *Andersen* [19] or *Steensgaard* [73]-style algorithm. The points-to information computed is flow-insensitive, and we store it as graphs. These function-wise pointer-assignment graphs (PAGs) are used to resolve potential call targets at dynamic call sites. We merge those intra-procedural PAGs later to obtain inter-procedural pointer information while constructing the callgraph. We use $\pi_{\texttt{C::f}}$ to denote a pointer-assignment graph for function f in module C. We use $\Pi_{\texttt{C}}$ to denote a pointer-assignment graph containing all pointer-assignment graphs for module C.

▶ **Example 2.** For each function definition contained in the `Sanitizer` module a PAG is computed and added to the graph $\Pi_{\texttt{Sanitizer}}$. The $\Pi_{\texttt{Sanitizer}}$ graph containing $\pi_{\texttt{Sanitizer}} :: {\sim}\texttt{Sanitizer}$, $\pi_{\texttt{Sanitizer}} ::$ sanitize , $\pi_{\texttt{Sanitizer}} ::$ isMalicious , and $\pi_{\texttt{Sanitizer}} ::$ applySanitizer is shown in Figure 5. Inter-procedural points-to relations are not followed and thus, formal pointer-typed parameters and calls to functions that return a pointer value remain unresolved and represent boundaries to the respective PAG. For instance, the pointer s in the applySanitizer() function points to the return value of getGlobalSan() which is indicated by a special node in the respective PAG (cf. Figure 5d).

3.2.3 Callgraphs and Inter-Procedural Points-To Information

After having computed the function-wise pointer assignment graphs, the callgraph is constructed according to Algorithm 1, Algorithm 2 and its resolver routine shown in Algorithm 3. The same algorithm also computes points-to information across procedure boundaries. Since one cannot know upfront what library functions a user is going to call, the callgraph algorithm has to consider every externally visible function definition as a possible entry point [54] (cf. line 58 of Algorithm 1). We use $CG_{\texttt{C}}$ to denote a (partial) callgraph of a module `C`. The algorithm starts at an arbitrary externally visible function f of module C. It then iterates through all call sites cs of f (cf. line 44). We denote a call site as cs_i where i represents the line number at which the call site is found. In the following, we write $\overline{cs_i}$ for a static call site and $\widetilde{cs_i}$ for a dynamic call site at which a function pointer or virtual function member is called. In case a static call site has been detected, the algorithm adds a new callgraph edge (line 46). In addition, for the pointer analysis, the algorithm connects the caller's actual

[2] Intra-procedural points-to information is, by definition, never invalidated by additional program information from other procedures.

pointer parameters and pointer return value with their corresponding formal parameters and return value of the callee target using a *stitch* operation (line 69), thus promoting (intra-procedural) pointer information to inter-procedural information. We formally define the stitch operation in Definition 3 and then discuss its use. In the latter case (line 48), the algorithm uses points-to information provided by Π_{C} to resolve potential call targets of $\widetilde{cs_i}$ according to Algorithm 3. Starting from the function pointer that is invoked or the pointer variable of the receiver that the virtual member function is being called on at $\widetilde{cs_i}$, we search in Π_{C} for reachable functions in case of function pointer calls (line 83) or allocation sites in case of virtual member function calls (line 96), respectively.

In this process, two situation may occur along with different levels of completeness of points-to information which dictate what (missing) dependencies must be tracked: *Incomplete or partially complete information:* If no functions or allocation sites are reachable yet, the reachable pointers at the function boundaries (i.e., formal pointer parameters or pointer return value of a function whose definition is missing) are marked as dependencies of $\widetilde{cs_i}$ (line 86 and 94). The dependencies are maintained in a bidirectional map from dependent pointer parameters to the respective unresolved call site and vice versa. If only some functions or allocation sites are reachable but also there are some reachable pointers at function boundaries as well, then pointers at function boundaries are added to the dependencies of $\widetilde{cs_i}$ and reachable functions are added as potential call targets to the callgraph (line 109 and 50). The edges of the callgraph are annotated with $\widetilde{cs_i}$. For virtual member function calls, the call targets of the allocated types at reachable allocation sites are inspected to find the potential targets (line 104) which are then added to the callgraph. *Complete information:* If no boundary pointers but only functions or allocation sites are reachable starting from the pointer at $\widetilde{cs_i}$, then no dependencies must be tracked.

During the construction of the callgraph we can have situations where a pointer-assignment graph will be amended with new information. To this end, we define a first graph operation which we call *stitch* and which we use to combine pointer information at call sites.

▶ **Definition 3.** *Stitch: Let $G = (V, E)$ be a (directed) graph containing vertices $\{u, v\} \subseteq V$ with $u \neq v$ and $e = (u, v) \notin E$. The stitch of u and v is a new graph $G' = (V', E')$, where $V' = V$ and $E' = E \cup (u, v)$. For convenience, we additionally define the function stitch : $G \times G' \times P \to G''$ that maps the (directed) graphs $G = (V, E)$ and $G' = (V', E')$, and P a set of pairs of vertices (u, v) with $u \in G$ and $v \in G'$ that shall be stitched together to a new graph G''. The stitch function $stitch(G, G', P)$ produces G'' such that $G'' = (V \cup V', E \cup E' \cup P)$.*

For each target function C::g that could be successfully resolved, the algorithm stitches $\widetilde{cs_i}$ to $\pi_{\mathsf{C::g}}$ (cf. line 69): Actual pointer parameters are connected with the corresponding formal parameters of the callee function C::g. If C::g returns a pointer parameter, it is connected as well. All edges are annotated with the corresponding call site.

If this graph stitch affects a pointer that is listed in the dependency map, the algorithm recursively continues resolving the affected call sites. Otherwise, the algorithm recursively continues resolving call sites in the resolved target functions. The algorithms for the interwoven points-to, callgraph computation are shown in Algorithm 1, Algorithm 2, and Algorithm 3. We use the symbol cs in a call to the function $stitch(G, G', cs)$ as shorthand for $\{(a_i, f_i)\}$, the set of pairs of left-hand-site pointer variable/actual pointer parameters and pointer return value/formal pointer parameters of the callee at cs that are stitched together.

▶ **Example 4.** The callgraph algorithm starts analyzing the function Sanitizer :: sanitize(). At the call site $\overline{cs}_{16}$, the actual parameter is stitched to the formal parameter of Sanitizer :: isMalicious() and the algorithm proceeds in Sanitizer :: isMalicious(). Since Sanitizer :: isMalicious() has now already been visited, the next function to be analyzed is applySanitizer().

Algorithm 1 Callgraph construction algorithm.

```
38 directed graph: CG_C = ∅, T = computeTypeHierarchy(); undirected graph: Π_C = ∅;
     bidirectional map: D = ∅; set: V = ∅;
39 Function constructionWalk(f):
40     if f ∈ V || isDeclaration(f) then
41         return;
42     end
43     V∪ = f;
44     foreach callsite cs ∈ f do
45         if cs is static then
46             CG_C∪ =< cs,getCallee(cs)>;
47             updatePointerInfo(f, getCallee(cs));
48         else
49             callees = resolveIndirectCallSite(cs);
50             foreach callee ∈ callees do
51                 CG_C∪ =< cs, callee >;
52                 updatePointerInfo(f, callee);
53             end
54         end
55     end
56     return;
57 Function constructCallGraph():
58     foreach f ∈ C do
59         if !isDeclaration(f) then
60             Π_C∪ = computePointsToGraph(f);
61     end
62     foreach f ∈ C \ {internalfunctions} do
63         if f ∉ V∧ !isDeclaration(f) then
64             CG_C∪ = f;
65             constructionWalk (f);
66     end
67     return;
```

Algorithm 2 Procedure for updating the pointer information.

```
68 Function updatePointerInfo(f, callee):
69     Π_C = stitch(Π_C[f], Π_C[callee], cs);
70     modptrs = getVerticesInvolvedInGraphOp(stitch, Π_C[f], Π_C[callee], cs);
71     foreach ptr ∈ modptrs do
72         if ptr ∈ D then
73             fmod =getFunctionContaining(D[ptr]);
74             V = V \ fmod;
75             constructionWalk(fmod);
76     end
77     constructionWalk(callee);
78     return;
```

applySanitizer() contains two interesting call sites. $\overline{cs}_{22}$ is a static call to getGlobalSan(). However, its definition is currently not available and thus, a callgraph node which is marked as a declaration is added to the callgraph. Note that the function causes incomplete points-to information as it returns a pointer value that is stored in variable s (cf. Figure 5d).

Furthermore, a virtual function member is called at $\widetilde{cs}_{23}$ on the receiver pointer variable s of type Sanitizer∗. Due to dynamic dispatch we have incomplete information on the possibly called functions and are not able to resolve this call, because we cannot yet determine the allocation sites that are reachable through s. The algorithm marks this call site as incomplete and keeps track of the dependent pointer variable s. The call site has to be updated as further information might be discovered later on. For instance, if the definition of getGlobalSan() becomes available that provides the required additional points-to information. The partial callgraph that can be computed individually on the `Sanitizer` module is shown in Figure 6.

Algorithm 3 Procedure for resolving dynamic call sites.

```
79  Function resolveIndirectCallSite(cs):
80      callees = ∅;
81      if isFunctionPtrCall(cs) then
82          fptr = getCalledPtr(cs);
83          rfptrs = getReachablePtrs(fptr);
84          foreach fptr' ∈ rfptrs do
85              if isBoundaryPtr(fptr') then
86                  D[cs] ∪= fptr';
87          end
88          callees ∪= getReachableFunctions(fptr);
89      else
90          aptr = getAllocationPtr(cs);
91          raptrs = getReachablePtrs(aptrs);
92          foreach aptr' ∈ raptrs do
93              if isBoundary(aptr') then
94                  D[cs] ∪= aptr';
95          end
96          allocs = getReachableAllocSites(aptr);
97          foreach alloc ∈ allocs do
98              τ = getAllocatedType(alloc);
99              v_τ = getVTable(T, τ);
100             if ! v_τ then
101                 D[τ] ∪= cs;
102             else
103                 i = getVCallIndex(cs);
104                 callee = getVTableEntry(v_τ, i);
105                 callees ∪= callee;
106             end
107         end
108     end
109     return callees;
```

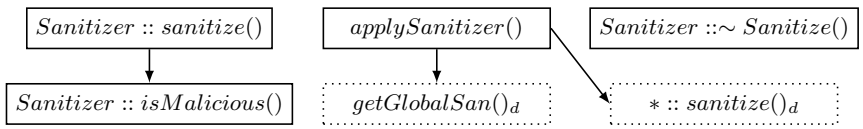


Figure 6 Callgraph for `Sanitizer`: $CG_{\texttt{Sanitizer}}$. f_d denotes the declaration of a function f.

3.2.4 Background on IFDS/IDE

To illustrate how ModAlyzer summarizes data-flow information, as foundational background we first present the inherently compositional *Interprocedural Finite Distributive Subset (IFDS)* [55] and *Interprocedural Distributive Environments (IDE)* [63] frameworks that ModAlyzer utilizes to solve data-flow problems.

The IFDS and IDE frameworks both follow the functional approach [66] to inter-procedural data-flow analysis. We use IFDS/IDE to encode our data-flow analyses as they allow the generation of graph-based, precise, reusable data-flow summaries of regions of code, even for incomplete code. Additionally, IFDS/IDE allow for the composition of data-flow summaries. This is required when implementing a compositional approach throughout all pieces of information where parts of the program are missing while the data-flow analysis is performed.

Reps et al. showed that distributive data-flow problems can be solved elegantly and efficiently by transforming them into graph reachability problems. The IFDS framework and its generalization IDE construct an exploded super-graph (ESG) in which each node represents a data-flow fact. If a data-flow fact d holds at a statement s the node (s, d) is

reachable in the ESG from a special tautological fact Λ (that always holds) and vice versa. The ESG is constructed by replacing each node in the inter-procedural control-flow graph (ICFG) with a bipartite graph that represents the equivalent flow function and thus, describes the effects of the statement on the data-flow facts. Standard functions for generating (Gen) or removing (Kill) data-flow facts can be encoded in bipartite graphs. Therefore, all Gen/Kill problems such as live variables, available expressions, etc. can be encoded within IFDS/IDE.

The runtime complexity of IFDS is $\mathcal{O}(|N| \cdot |D|^3)$, where $|N|$ is the number of nodes in the ICFG and $|D|$ is the size of the data-flow domain D. Thus, the efficiency highly depends on the size of the underlying data-flow domain.

In IDE, however, the data-flow domain D is decomposed into the data-flow domain D and a separate value domain V. The value domain V can be infinite. Because IDE has the same complexity as IFDS, the size of V does not affect the complexity of the algorithm. IDE annotates the edges of the ESG with lambda-functions that describe a value computation over the domain of V. When a reachability check is performed in IDE to decide whether an ESG node (s, n) is reachable and, therefore, the fact d holds at statement s, the value computation problem that is specified along those edges leading to (s, d) is solved. Figure 7 shows some exploded super-graphs for a taint analysis conducted on the code in Listing 2.

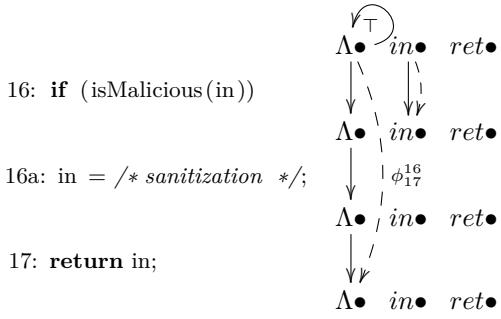
IFDS and IDE follow the functional, summary-based approach to achieve fully context-sensitive, inter-procedural analysis. The effect of statements of sections of code can be summarized by composing the flow functions of subsequent statements. The composition $h = g \circ f$ of two flow functions f and g, called *jump function*, can be obtained by combining their bipartite graph representations. The graph of h can be produced by merging the nodes of g with the corresponding nodes of the domain of f. Once a summary ψ for a complete procedure p has been constructed, it can be (re)applied in each subsequent context the procedure p is called. Importantly, because the flow functions are assumed to distribute over the merge operator, this summarization is known to be *lossless* [55]. IFDS/IDE problems can thus be solved with full precision, without the need for approximation.

3.2.5 Data-Flow Information

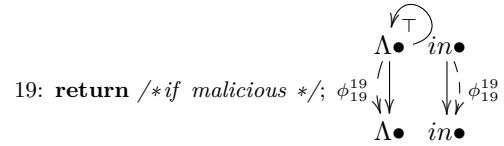
In the next step, the analysis computes the possibly partial data-flow information using IFDS/IDE [51, 55, 63] according to the description of the data-flow problem to be solved for the available function definitions. In contrast to the information computed before, the data-flow information depends on the configuration of the client analysis because data-flow information is never general and always depending on a specific definition.

We use the flow and edge functions of the client analysis to construct the partial exploded super-graph of the library to be summarized. The partial callgraph is traversed in a depth-first bottom-up manner to maximize the number of functions that can be summarized completely. For a library function f that does not make any calls, the summary information is computed by applying the client's flow and edge functions to each node n of the control-flow graph. The resulting exploded super-graph edges are then combined using composition and meet to construct the *jump functions* $\psi(f)$ that summarize the complete function. For each incoming data-flow fact d_i, its respective jump function $\phi_i(f)$ describes the effect of the analyzed function on d_i.

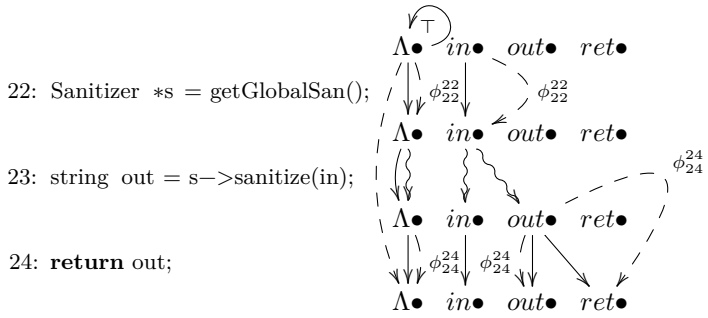
In case a function f contains call sites cs_i, the IFDS/IDE algorithm computes a partial data-flow summary from f's entry node to the first call-site node cs_1, $\psi_{cs_1}^{entry}(f)$. It then computes the summary of the called function f', $\psi(f')$, (if not already computed) and composes it with the partial summary $\psi_{cs_1}^{entry}(f)$ to obtain $\psi_{rs_1}^{entry}(f)$. The algorithm proceeds successively until the complete summary $\psi(f) = \psi_{exit}^{entry}(f)$ has been constructed.



(a) Exploded super-graph for Sanitizer :: sanitize().



(b) Exploded super-graph for Sanitizer :: isMalicious().



(c) Exploded super-graph for applySanitizer().

Figure 7 Exploded super-graphs for the `Sanitizer` module.

However, in case a library function f contains call sites that are depending on user code, for instance, because of callbacks or incomplete points-to information, a complete summary $\psi(f)$ cannot be computed. In this case, ModAlyzer computes a set of partial summary functions ψ^n_m, where n is a function's entry point or some return site (rs) and m is a function's exit statement or some call site (cs) whose call targets are not or only partially known. This results in *gaps* in the exploded super-graph that represent the unresolved effects of the missing call targets.

▶ **Example 5.** The data-flow information computed for the `Sanitizer` module is shown in Figure 7. Individual flow/edge functions are denoted by solid ($\rightarrow$) and jump functions by dashed ($\dashrightarrow$) arrows. Analyzing applySanitizer() leads to an incomplete ESG, because the callgraph for the `Sanitizer` module is only partially complete. The definition of getGlobalSan() is not yet available and the dynamic call site at line 23 cannot be resolved with the information available within the `Sanitizer` module.

The call to the unresolved function getGlobalSan() does not interact *directly* with the data-flow information as it receives no arguments, its return type differs from the type of the data-flow domain (strings), and the string which the variable in refers to is not global as no global declarations are present. Therefore it cannot be modified by the call and one can safely use the identity function here. We will further elaborate on that in Subsection 3.4.

The call to $*$::sanitize() results in a *gap* in the ESG. In Figure 7c gaps in the ESG are indicated with squiggled arrows ($\rightsquigarrow$). We pass *in* and *out* as identity after the gap and also generate other variables, such as the implicit return variable, that depend on out. Later

on, after the merging process, the missing targets of the call site at line 23 will have been determined and their data-flow summaries can be inserted. Then, the analysis will check whether *in*, *out*, and *ret* are reachable from Λ, and determine if those variables are tainted.

The ESGs for the Sanitizer :: sanitize () and Sanitizer :: isMalicious () functions are shown in Figure 7a and 7b, respectively. For our example analysis we assume that Sanitizer :: isMalicious () checks whether the variable in contains malicious data and the function does not modify the data-flow facts. Sanitizer :: sanitize () checks if the string referred to by variable in contains malicious data – is tainted – and, if so, replaces it with a sanitized version. Again, to keep our example analysis simple, we assume that the analysis is aware of the special semantics of Sanitizer :: isMalicious () and thus, kills the variable in in both branches.

After having computed the data-flow summaries for the `Sanitizer` module, we have determined any information we need on `Sanitizer` as an individual module. We denote the combination of the partial type-hierarchy graph (and call targets) in Figure 4, partial points-to in Figure 5 and callgraph in Figure 6 and the partial data-flow summaries for `Sanitizer` in Figure 7 as $\Xi_{\texttt{Sanitizer}}$.

3.3 Merging Analysis Summaries

To complete the picture, we next combine the information obtained by analyzing `Sanitizer` and `DbgSanitizer` with an analysis of the client application `Main`.

For this we need to define a new operation on graphs which we call *contraction*. We use the contraction operation when new information becomes available during a merge, to replace placeholder nodes (that indicate missing information) of a graph by their counterparts that represent the actual information. We apply this operation to combine partial type hierarchy- and callgraphs. For instance, we combine callgraphs by *contracting away* function declaration nodes with their respective definition counterpart nodes: the nodes representing function declarations are removed and all former incoming edges now lead to the corresponding definition nodes. We formally define the contraction operation as follows:

▶ **Definition 6.** *Contraction: Let $G = (V, E)$ be a (directed) graph containing vertices $\{u, v\} \subseteq V$ with $u \neq v$. Let f be a function that maps every vertex in $V \setminus \{u, v\}$ to itself, and otherwise, maps it to a new vertex w. The contraction of u and v is a new graph $G' = (V', E')$, where $V' = (V \setminus \{u, v\}) \cup \{w\}, E' = E \setminus \{e = (u, v)\}$, and for every $x \in V$, the vertex $x' = f(x) \in V'$ is incident to an edge $e' \in E'$, iff the corresponding edge $e \in E$ is incident to x in G (reproduced from [53]). For convenience, we additionally define the function $contract : G \times G' \times P \to G''$ that maps the (directed) graphs $G = (V, E)$ and $G' = (V', E')$, and P a set of pairs of vertices $u \in V$ and $v \in V'$ that shall be contracted to a new graph G''. The contraction function $contract(G, G', P)$ contracts the pairs of vertices u_i and v_i and produces a new (directed) graph $G'' = ((V \cup V') \setminus \{u_i\}, (((E \cup E') \setminus \{(t_j, u_i)\}) \setminus \{(u_i, v_i)\}) \cup \{(t_j, v_i)\})$, where all edges incident to u_i with their origin in some vertex t_j are replaced by edges from t_j to v_i contracting away u_i. We use f in $contract(G, G', f)$ as shorthand for $\{(f_{decl}, f)\}$, the set of function declaration/definition pairs and τ in $contract(G, G', \tau)$ as shorthand for $\{(\tau_{decl}, \tau)\}$, the set of type declaration/definition pairs.*

Our merge procedure for two module summaries Ξ_i and Ξ_j is shown in Algorithm 4. In the following, we present all involved steps for each piece of analysis information.

3.3.1 Type Hierarchies

The analysis first merges the type-hierarchy graphs using vertex contraction (cf. line 114), to remove redundant definitions of the same type. The redundancy is caused by including a type's definition (which usually resides in a corresponding header file) in multiple modules that require a type's exact data layout (e.g. for allocation or subtyping).

▶ **Example 7.** While performing the contraction, the analysis finds that `Sanitizer`'s type $\tau_{\texttt{Sanitizer}}$ is sub-typed by $\tau_{\texttt{DbgSanitizer}}$. The contraction has no immediate effect on the callgraph analysis: As the callgraph uses points-to information to resolve indirect calls, no immediate update is required at this point, because the new type-hierarchy information is not used before new pointer information becomes available. The type hierarchy needs to be queried if a new allocation site has been found. For each newly discovered allocation site, the type hierarchy is used to retrieve the entry of the allocated type's virtual function table.

3.3.2 Callgraphs and Points-To Information

The analysis merges the callgraphs by using the vertex contraction operation introduced before (line 122). A contraction is used to remove function-declaration nodes and replace them with their corresponding definition nodes, now linking calls to callees. While performing the contraction on the callgraphs, the corresponding partial pointer-assignment graphs are not contracted but stitched together (cf. Definition 3); through the stitch (line 126) no nodes of the pointer-assignment graph are replaced to keep information on the parameter mapping. Actual pointer parameters at a call site as well as pointer return values at the respective return site are connected with the corresponding formal parameters of the called function and the left-hand side variables, respectively. The information on the contracted callgraph nodes is used in the next step when *repropagating* data-flows.

Algorithm 4 Merge procedure for callgraphs.

```
110 Function merge(CG_C, T_C, Π_C, D_C, V_C, CG_C', T_C', Π_C', D_C', V_C',):
111     D_C ∪= D_C';
112     V_C ∪= V_C';
113     Π_C ∪= Π_C';
114     T_C = contract(T_C, T_C', τ);
115     modtypes = getVerticesInvolvedInGraphOp(contract, T_C, T_C', τ);
116     foreach τ ∈ modtypes do
117         if τ ∈ D then
118             f =getFunctionContaining(D[τ]);
119             V = V \ f;
120             constructionWalk(f);
121     end
122     CG_C = contract(CG_C, CG_C', f);
123     {⟨cs, f⟩} = getVertexPairsInvolvedInGraphOp(contract, CG_C, CG_C', f);
124     foreach ⟨cs, f⟩ do
125         f' =getFunctionContaining(cs);
126         Π_C = stitch(Π_C[f'], Π_C[f], cs);
127         modptrs = getVerticesInvolvedInGraphOp(stitch, Π_C[f'], Π_C[f], cs);
128         foreach ptr ∈ modptrs do
129             if ptr ∈ D then
130                 f =getFunctionContaining(D[ptr]);
131                 V = V \ f;
132                 constructionWalk(f);
133         end
134     end
```

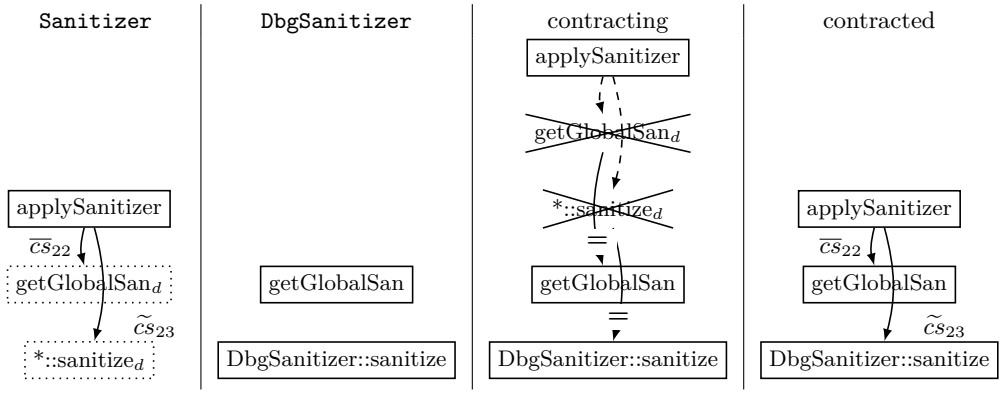


Figure 8 Excerpt of the vertex contraction for callgraphs of `Sanitizer` and `DbgSanitizer`. f_d denotes the declaration of a function f.

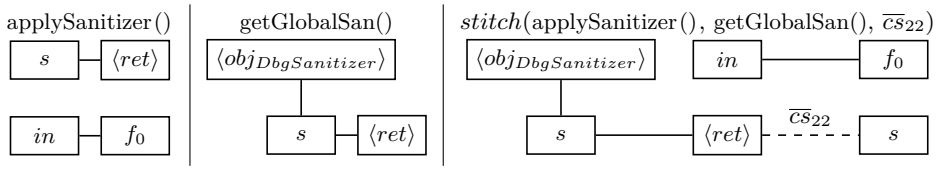


Figure 9 Excerpt of the vertex stitch of the PAG's for applySanitizer() and getGlobalSan().

▶ **Example 8.** The callgraph contraction of the modules `Sanitizer` and `DbgSanitizer` is indicated in Figure 8. The callgraph contraction triggers the corresponding stitching of PAGs. For instance, the points-to graphs $\pi_{\texttt{Sanitizer::applySanitizer}}$ and $\pi_{\texttt{getGlobalSan}}$ are stitched together at $\overline{cs}_{22}$ as indicated in Figure 9. Through the stitch, the analysis recognizes that the previously marked pointer variable s gets new inputs from the resolved callee function getGlobalSan(). As s is now able to reach getGlobalSan()'s variable s of allocated type $\tau_{\texttt{DbgSanitizer}}$ and the receiver object s in applySanitizer() has no other unresolved dependencies, the possible call targets are updated in the callgraph such that DbgSanitizer::sanitize() is now the only possible target for the dynamic call site at line 23. The pointer-assignment graph of the newly discovered callee at line 23 is stitched to the call site $\widetilde{cs}_{23}$.

3.3.3 Fixed-Point Iteration for Callgraph and Points-To Graph

Note that there are cases in which the stitch (of two PAGs) of a resolved callee function changes the points-to information in such a way that previously partially resolved indirect call sites must be revised again (cf. line 69 for summarization, and line 126 for merges). In these cases, the analysis loops in updating callgraph and points-to information until the callgraph and points-to information stabilize. A constructed yet expressive example of the

```
void (*f)();                                                              131
void bar() {}                                                             132
void foo() { f = &bar; }                                                  133
void init(void (*f)()) { f = &foo; }                                      134
int main() { init(f); f(); /* <-- indirect call site */ return 0; }       135
```

Listing 4 Example in which the update of points-to- invalidates callgraph information.

aforementioned for function pointers is shown in Listing 4. When the callgraph algorithm resolves the indirect call to the function pointer f using points-to information, it determines foo() as the callee target. However, foo() manipulates the points-to information such that bar() becomes a feasible target as well. Thus, the indirect call site has to be revisited and bar() has to be added as a possible target as well. When analyzing bar() the callgraph and points-to information stabilize and the algorithm terminates.

3.3.4 Data-Flow Information

Once a callgraph has been updated by a merge, the data-flow information has to be repopulated in order to reflect the changes. Whenever two callgraphs are merged, new function definitions and their respective data-flow summaries become available which have been previously unknown to the other module's data-flow information. The merge procedure for the callgraphs shown in Algorithm 4 issues the contracted nodes (function declarations) and their respective call sites. This information and the newly available function definitions and accompanying data-flow summaries are used to close potential gaps in the ESG. The analysis visits all sub-graphs that have undergone the callgraph contraction procedure in a depth-first bottom-up manner, filling in the newly available data-flow summaries.

Suppose a function f contains a previously unresolved or only partially resolved call site cs and therefore, a pair of partial summaries $\psi_{cs}^{entry}(f)$ and $\psi_{exit}^{rs}(f)$. If the callgraph contraction reveals the call target f' and its respective data-flow summary, $\psi_{cs}^{entry}(f)$ and $\psi_{exit}^{rs}(f)$ are composed with $\psi(f')$ to produce a complete summary of f, $\psi_{exit}^{entry}(f) = \psi_{exit}^{rs}(f) \circ \psi(f') \circ \psi_{cs}^{entry}(f)$. The summary $\psi_{exit}^{entry}(f)$ may need to be merged with any existing jump functions that have been obtained along other paths, for instance, call-free-paths (cf. flows for Λ in Figure 7c) or paths for other call targets of cs that have been available for analysis already. The complete summary $\psi(f)$ is used to successively fill in potential other gaps in the ESG.

In case a target library to be summarized is depending on code of its user(s) because it uses features such as callbacks, for instance, the static analysis summaries Ξ even for the complete library code will contain gaps. Those gaps are eventually closed once the main application is available, analyzed and merged with the precomputed library summaries to produce the final analysis results.

▶ **Example 9.** As the function definition of DbgSanitizer::sanitize() becomes now accessible to applySanitizer(), its respective data-flow summary can now be plugged into the current gap of applySanitizer() to obtain a complete IFDS/IDE summary for it. The sub-graphs that undergo the contraction procedure are visited in a depth-first, bottom-up manner and the data-flow summary for DbgSanitizer::sanitize() is inserted into applySanitizer(). The analysis therefore finds that the values passed as a reference parameter into DbgSanitizer::sanitize() and the value returned by it are indeed tainted. Therefore, the return value of applySanitizer() is tainted as well. The pre-analysis of the library is now complete and the obtained results can be used by any potential client to the library.

3.3.5 Analyzing the Main Application

When analyzing the application program `Main` the analysis first constructs `Main`'s type hierarchy, function-wise pointer-assignment and callgraph (cf. Algorithm 1). The type hierarchy-, call- and pointer-assignment graphs for `Main` are merged with the library's respective graphs (cf. Algorithm 4). The data-flow analysis can then start at the entry point

main(). As the data-flow analysis recognizes the call to applySanitizer() it can directly use the (complete) pre-computed summary and thus keeps the return value as well as the actual reference parameter input marked as tainted. Finally, the client analysis is able to query the results and finds that the tainted variable sanin leaks at the call to Statement::executeQuery().

3.4 Removing Dependencies Ahead of Time

While computing the data-flow information for an individual module, information at dynamic call sites or static call sites, where the callee definitions are not available, will be incomplete. However, by using the following shortcuts, ModAlyzer is able to compute a complete and precise data-flow summary nonetheless. We already observed such a situation while computing the data-flow information for applySanitizer() in the `Sanitizer` module. Because the call to getGlobalSan() at line 22 does not have a *direct* impact on the data-flow information (as described in Example 5), we can model it using the identity flow function. Note, however, that the call still has an *indirect* impact since the function is able to change what function is being called in the next line. When our analysis recognizes a function f that misses information on potential callees, but where we can ensure that the missing information has no direct or indirect impact on the *data-flow* information, we can nevertheless compute a complete and precise summary for f using the *identity* shortcut denoted as $\overset{\text{id}}{\hookrightarrow}$ and thus fully remove any dependencies on the missing callees. To determine if $\overset{\text{id}}{\hookrightarrow}$ can be applied, different predicates may be applied, depending on the client analysis, e.g. *pass and return by value*. For instance, if a function receives its arguments *by value* they are copied into the callee. Thus, we can be sure that it cannot modify its arguments even if information on the callee's definition is missing.

```
string foo(bool p){string in = userInput(); return p ? sanitize(in) : in;}
```

Listing 5 Code allowing the $\overset{\top}{\hookrightarrow}$ *shortcut*.

Another example of a situation in which a data-flow analysis can perform such an optimization is shown in Listing 5. Such a treatment for summarization of incomplete data-flow analysis has also been presented in [43]. While analyzing foo() we assume the information $\top$ for the variable in, i.e., in is tainted. foo() sanitizes in only in one of the branches (depending on an unknown predicate). Hence, if we assume that we are conducting a may-taint analysis, then it holds that in *may* be tainted at the end of foo() *no matter what* the call to sanitize() does. It follows that $\top$ will always be associated with in. In this case, we can compute a complete summary even with incomplete information by using the $\top$ shortcut $\overset{\top}{\hookrightarrow}$. This is always true for *may*-analyses that use set union as the merge operator, which for instance in IFDS is always the case.

In the presence of global variables, ModAlyzer applies shortcuts *only* if they can be proven sound, which ModAlyzer manages easily if only module-internal global variables are involved. Global variables are often declared as static (in case of C) or within anonymous namespaces (in case of C++) making them internal to the module that declares them. ModAlyzer's shortcuts are not applied if externally visible global variables are involved in the situation, i.e., variables that are used across multiple modules.

Due to C/C++'s modular compilation model, an analysis frequently encounters situations as presented above, in which it can use these shortcuts to compute data-flow information. Functions where these shortcut summaries are used do not need to be revisited, thus, the analysis is able to work more efficiently. Therefore, when summarizing a module, it is desirable to remove as many data-flow dependencies as possible using the $\overset{\text{id}}{\hookrightarrow}$ and $\overset{\top}{\hookrightarrow}$ shortcuts.

4 Implementation

We have implemented the strategy described in Section 3 in a tool called ModAlyzer, as an extension to PhASAR [64], a static-analysis framework that has been implemented on top of LLVM [45]. PhASAR allows to solve arbitrary monotone data-flow problems on the LLVM intermediate representation (LLVM IR) and also provides IFDS/IDE solver implementations.

We extended the existing IDE solver as well as the other infrastructure for type hierarchy, points-to, and callgraph computation and added the necessary summarize, merge, and update functionalities respectively.

ModAlyzer persists the summary results by using a document-oriented store in which it saves the graphs along with the code the analysis is conducted on with help of LLVM's metadata capabilities. LLVM allows for a key-based introduction of custom metadata. Each function that is defined in a module is annotated with its function-wise summaries for the different pieces of static analysis information, i.e., its points-to and exploded super-graph. A module carries the module-wise information that is obtained by merging all information of its enclosed functions as well as type hierarchy and callgraph information. Those module-wise summaries are referred to using the module flags section of the LLVM IR.

For the persistence, we created a bidirectional mapping from LLVM's in-memory representation to a textual representation allowing us to store the graphs comprising pointer values to LLVM IR records as graphs that use the text-encoded version. Additionally, we implemented *import* and *export* functionalities for each graph type that enable us to manage loads and stores of encoded graphs along with the LLVM IR.

LLVM's metadata mechanism does not restrict the type of data for annotations. Thus, arbitrary data structures and encodings may be used to persist the analysis information. We use the capabilities of the Boost Graph Library (BGL) [67] to manage type hierarchy, points-to, and callgraph information. The BGL offers of-the-shelf textual import and export functionalities and allows for implementing custom reader/writer concepts. We use the default Graphivz [15] format to store the graphs in metadata records. As PhASAR's IFDS/IDE solver implementation works by incrementally constructing two tables to represent flow functions/jump functions of ever longer sequences of code (c.f. [51,64]), we use the following sets of quintuples for the data-flow summary representation of a function $\psi(f) := \{\langle n_i, d_x, n_j, d_y, l\rangle\}$, where a quintuple represents a jump function (or an edge in the ESG) from data-flow fact d_x to d_y with the corresponding edge function l that summarizes parts of the effects of the region of code that is enclosed by the statements n_i and n_j. The concrete (partial) data-flow summary for the applySanitizer() function (cf. Figure 7c) looks as follows: $\{\langle 22, \Lambda, 24, \Lambda, \top\rangle, \langle 22, in, 22, in, \top\rangle, \langle 24, in, 24, in, \top\rangle, \langle 24, out, 24, out, \top\rangle, \langle 24, out, 24, ret, \top\rangle\}$. Note that for IFDS we can use the simple encoding of the binary lattice and the edge functions. We handle the persistence of the difficult-to-handle, general IDE edge functions by creating a record to keep track which edge functions are composed and meet for each jump function while constructing them. We finally persist the record using the extensive Boost Serialization library [14]. On load, the record can be *replayed* to (re)construct the actual jump functions.

5 Experiments

Our empirical evaluation aims to answer the following research questions:

- **RQ1**: Does the use of a module-wise static analysis incur a precision loss when compared to a whole program analysis? If so, what causes this loss in precision?

- **RQ2**: Compared to conducting a whole-program analysis, what speed-up can one achieve when applying MWA using pre-computed summaries for type-hierarchy, callgraph, points-to and data-flow information?
- **RQ3**: How frequently can the data-flow shortcuts $\overset{\mathsf{id}}{\hookrightarrow}$ and $\overset{\top}{\hookrightarrow}$ be applied in MWA?

To address **RQ1**, we compare the analysis results of a whole program analysis with the results obtained by a module-wise analysis. Ideally, the results of both analyses should be identical. To address **RQ2**, we measure and compare the runtimes of a client analysis using pre-computed summaries and a version that computes everything on-the-fly. To address **RQ3**, we extend PhASAR's IFDS/IDE solver implementation and measure how frequently it makes use of both shortcuts for different client analyses.

5.1 Experimental Setup

We have evaluated ModAlyzer using as benchmark subjects the C coreutils (version 8.28) [3] and the PhASAR framework itself.

The GNU core utilities are a collection of C programs that share a common core, providing a library that consists of 251 files. Each coreutil program itself only consists of a small number of C source files that provides the program's entry point, manages the command-line, and makes suitable calls into the common core in order to achieve the desired task. For our evaluation we prepared and analyzed 97 of the coreutils and chose 10 of them at random which to present in this paper in more detail. (However, the figures for the remaining 87 coreutils can be found online [16].)

PhASAR is written in C++ and is similarly structured. To provide flexible, reusable software components, the main functionalities of the different components are implemented as libraries. The front-ends (or drivers) themselves represent only a relatively small amount of "glue code" and large amounts of their runtime is spent in library code. Using PhASAR we defined two benchmark subjects: First PhASAR's own command-line client and the PhASAR-based tool *MPT*, a exemplary client that uses PhASAR as a library, both of which can be found alongside PhASAR's examples [10].

We chose those subjects because they have a relatively high amount of virtual calls. This stresses ModAlyzer's points-to based callgraph algorithm. We observed that C++ developers generally try to minimize the amount of indirect calls to avoid indirect jumps, which degrade performance, especially when implementing performance critical software systems [17]. The chosen subjects hence set a relatively high bar when it comes to evaluating analysis performance. The raw as well as the processed data produced in our evaluation is available online [16].

All programs and their characteristics are shown in Table 1. We prepared all programs presented for analysis with the PhASAR framework by compiling them into LLVM IR with production flags using the Clang compiler. The numbers in Table 1 are based on LLVM IR.

We used an uninitialized-variables analysis $\mathbb{U}$ and a taint analysis $\mathbb{T}$ as two concrete client analyses that both impose the information dependencies as shown in Figure 3. $\mathbb{U}$ and $\mathbb{T}$ are both implemented in IFDS within PhASAR.

Uninitialized-variables analysis $\mathbb{U}$: $\mathbb{U}$ is an analysis that finds potentially uninitialized variables and tracks them through the program. If the analysis finds an uninitialized variable to be read from, it reports an *illegal* use of that variable. Uninitialized variables propagate through computations and thus, the analysis tracks those as well. $\mathbb{U}$ also tracks the variables across function boundaries making it an inter-procedural analysis.

Table 1 Number of compilation units, library/application code ratio, number of statements, pointer variables and allocation sites of the analyzed (completely linked) programs.

Program	Compilation Units	$\frac{IR\ LOC\ lib}{IR\ LOC\ app}$	Statements	Pointers	Allocation Sites
wc	252	41.2	63,166	10,644	396
ls	253	5.9	71,712	13,200	438
cat	252	66.3	62,588	10,584	391
cp	256	10.5	67,097	11,722	443
whoami	252	335.7	61,860	10,433	389
dd	252	16.8	65,287	11,150	408
fold	252	105.8	62,201	10,509	390
join	252	24.9	64,196	11,042	402
kill	253	88.2	62,304	10,527	394
uniq	252	60.1	62,663	10,650	396
MPT	156	13.8	1,351,735	755,567	176,540
PhASAR (driver)	156	56.4	1,368,297	763,796	178,486

Taint analysis $\mathbb{T}$: $\mathbb{T}$ is a parameterizable taint analysis that tracks tainted values through the program and reports potential leaks whenever it finds a tainted value that may flows into a *sink* function (or operation). *Sources* and *sinks* are parameterizable. We used PhASAR's default parametrization that treats the command-line arguments passed into `main` as tainted. All standard *input* functions (e.g., `fread()`, `fgets()`) are treated as *sources* as well. All *output* functions (e.g., `fwrite()`, `printf()`) are treated as *sinks*.

For each target program shown in Table 1 we computed the library and application code ratio based on lines of LLVM IR code. If a module is used by more than one application, we consider it to be part of the library, whereas modules that are only used by one application are considered as application code. We also measured runtimes and number of leaks/uninitialized variables that each of the analyses reported in a WPA setup as well as an MWA setup. The measurements for MWA are split into a summarization and an actual analysis step. The PhASAR framework implements a reporting system which we use to compare the actual reports to make sure that the findings are identical. We also recorded the number of callgraph updates $\#CG\circlearrowleft$ that had to be performed in the MWA setup, i.e., we counted the number of callgraph edges that have been introduced during the merge process. This is a good indicator of the expense of a merge, as the introduction of a new callgraph edge causes the points-to and data-flow information to be updated as well. In addition, we measured the number of shortcuts that a data-flow analysis was able to use. We measured the runtimes by performing 5 runs for each analysis in each setup on a virtual machine running on an Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz machine with 128GB memory. We removed the minimum and maximum values and computed the average of the remaining 3 values. Table 2 shows the results. The first column comprises the programs under analysis, the second column contains the WPA runtimes, column three contains the required runtime for summarization, column four the actual analysis time of MWA. The differences of the runtimes and reports of WPA and MWA are shown in column five. Column six, seven, and eight contain the respective number of callgraph updates, identity shortcuts, and $\top$ shortcuts, respectively. The number of callgraph updates are equal for both analysis as the callgraph information is not affected by the concrete client analyses.

Table 2 Runtimes and findings WPA vs. MWA for the taint analysis $\mathbb{T}$ (first half) and uninitialized variables $\mathbb{U}$ (second half).

$\mathbb{T}$: Program	WPA [s]	$\Sigma_{m \in lib}$ [s]	MWA [s]	Δ runtimes / (Δ reports)	$\#\circlearrowleft^{CG}$	$\#\hookrightarrow^{id}$	$\#\hookrightarrow^{\top}$
wc	2.3	5.7	0.5	-1.8 / (0)	47	8,052	78
ls	4.8	5.7	1.3	-3.5 / (0)	166	13,470	11
cat	1.9	5.7	0.2	-1.7 / (0)	21	2,117	269
cp	4.4	5.7	1.8	-2.6 / (0)	197	19,712	1077
whoami	2.0	5.7	0.4	-1.6 / (0)	4	6,065	11
dd	8.1	5.7	5.5	-2.6 / (-3)	58	48,747	90
fold	2.1	5.8	0.4	-1.7 / (0)	12	6,695	11
join	2.4	5.7	0.6	-1.8 / (0)	58	8,979	11
kill	1.9	5.7	0.2	-1.7 / (0)	14	2,079	11
uniq	2.2	5.7	0.4	-1.8 / (0)	29	7,281	11
MPT	2,306	42,847	1,516	-809 / (0)	41	29,061	0
PhASAR	7,176	42,876	598	-6578 / (0)	3	47,736	0
$\mathbb{U}$: Program	WPA [s]	$\Sigma_{m \in lib}$ [s]	MWA [s]	Δ runtimes / (Δ reports)	$\#\circlearrowleft^{CG}$	$\#\hookrightarrow^{id}$	$\#\hookrightarrow^{\top}$
wc	2.6	5.9	0.6	-2.0 / (0)	47	2,413	162
ls	8.4	6.0	3.3	-5.1 / (0)	166	7,173	184
cat	2.0	6.0	0.3	-1.7 / (0)	21	845	12
cp	5.2	5.9	2.2	-3.0 / (0)	197	6,684	1122
whoami	2.0	5.9	0.3	-1.7 / (0)	4	535	0
dd	3.1	5.9	0.9	-2.2 / (0)	58	2,522	16
fold	2.1	6.0	0.4	-1.7 / (0)	12	895	0
join	2.8	6.0	0.5	-2.3 / (0)	58	2,582	171
kill	2.2	6.0	0.4	-1.8 / (0)	14	793	12
uniq	2.5	5.9	0.5	-2.0 / (0)	29	1,433	17
MPT	3,811	53,703	2,958	-826 / (0)	41	137,722	8,136
PhASAR	10,160	53,348	968	-9,192 / (0)	3	210,032	24,446

5.2 RQ1: Precision

As the points-to and therefore, call- and control-flow graphs guide an analysis through a program, they may heavily influence the reported results. Therefore, we compared the callgraph obtained in an MWA setting with the one obtained in a WPA setting. We found that the callgraphs only differ at call-sites at which a static function pointer is called. In those cases, our MWA callgraph implementation turns out to be *more* precise as it does not consider every function of the complete program that matches the pointer's signature as a possible target, but only the ones reachable within the module whose address can actually be taken.[3] This reduces the number of infeasible call targets while retaining soundness.

We compared the client analyses precision and recall of WPA and MWA using PhASAR's reporting capabilities. Column Δ in Table 2 shows how many result entries differ from a WPA to an MWA setup for each client analysis. We only observed a difference in the reports for the "dd" program while performing the taint analysis. In this case, the analysis in WPA mode reports three leaks in a library function f_L, whereas the analysis in MWA reports none. We investigated the cause of this difference and found that this is actually a false positive in the WPA. The leaking function f_L is not called within the "dd" program. However, "dd" defines a static global function pointer p in the application code and the WPA analysis safely

[3] Reducing the set of feasible function pointer targets in WPA mode can be easily implemented.

assumes that f_L, which matches the function pointers signature, might be called. When the application code that defines the static function pointer is analyzed in MWA mode, the analysis does not find a declaration of f_L within the application code and therefore, its address cannot possibly be taken, preventing it to be a callee target of p. While one could adapt the WPA to be equally precise, the MWA obtains this precision automatically.

Since ModAlyzer does not need to overapproximate information it does indeed also preserve recall. The ModAlyzer approach has been designed to obtain this property by construction. Besides the differing result entries that are caused by the differences in the callgraph, both the results of ModAlyzer and WPA coincide.

The module-wise analysis generally yields the same precision as the whole-program analysis, in some cases even exceeds it.

5.3 RQ2: Performance

Table 1 shows that the library/application ratio ranges from 5.9 to 5675.6 and therefore, that the actual application code only comprises a small fraction of the complete program. One expects the MWA runtime to pay off better with increasing code ratios, since more pre-computed summaries can be (re)used for a program's library parts. The runtimes of both analyses measured in the WPA and MWA setup live up to that expectation. Looking at the programs with an especially advantageous library/application ratio such as whoami, fold, kill, cat, PhASAR, the use of pre-computed summaries saves between 81% and 91% of the analysis time. On average, MWA saves 72% of analysis time compared to WPA while MWA's initial one-time summarization step is, on average, 3.67 times as expensive than the corresponding run in a WPA setup. Thus, computing the initial summarization of the library (or infrequently changing) parts of a program is more expensive that performing a whole program analysis. Computing summaries will always be more expensive compared to computing plain WPA due to the additional overhead required for organizing and maintaining the summaries. In addition, many of PhASAR's critical analysis parts have undergone tremendous amounts of manual optimization while ModAlyzer's implementation for summary generation has not yet been optimized manually. As a concrete example, analyzing PhASAR in an MWA setup outperforms WPA with the seventh run using the taint analysis and after the sixth run for the uninitialized variables analysis – assuming an initial summary must be computed and no changes in PhASAR's library occur after summarization. For the MPT program, that has a larger number of callgraph updates to be performed, MWA pays off with the 54th run for the taint analysis and 64th run for the uninitialized variables analysis, respectively.

In case of PhASAR, runtime savings of 92% can be achieved as the application merely consists of few calls into the library code. This is underlined by the three callgraph updates that are necessary. We manually inspected the program and confirmed that, although the amount of front-end code is certainly large, it performs only very few calls into the corresponding library. A controller class, which is part of the library, is used to dispatch the different tasks to solve into calls to the adequate library functionalities. This shifts large parts of the computation to the offline MWA summarization phase.

The size of the persisted summaries that are stored along with the library code increase a library, on average, by a factor of five in size. The code and summaries for PhASAR require approx. 2.8 GB of memory for persistence and 30 MB for the core utils.

Summaries for static callgraph, points-to and data-flow analysis can be used to capture the analysis effects of libraries. After a one-time pre-computation effort, this allows a runtime reduction of 72%, on average, compared to the runtimes in whole-program mode.

5.4 RQ3: Shortcuts

The number of $\overset{\text{id}}{\hookrightarrow}$ shortcuts taken by an analysis is parameterized by a predicate as described in Subsection 3.4. For the analyses $\mathbb{U}$ and $\mathbb{T}$ we used the predicate *return type is void and uses pass-by-value*. However, different predicates might be useful for other analyses, depending on the specific assumptions that can be made on an analysis's domain. The results in Table 2 show that both shortcuts can be frequently applied during analysis. The $\overset{\text{id}}{\hookrightarrow}$ shortcut can be applied between 535 and 210,032 times depending on the client data-flow analysis that is performed. The $\overset{\top}{\hookrightarrow}$ shortcut can be applied between 0 and 24,446 times. We are confident that the number of $\overset{\top}{\hookrightarrow}$ shortcuts could be further increased, if one adjusts PhASAR's data-flow solvers to favour analyzing branches first that contains fewer (or no) function calls.

Shortcuts can be frequently applied. Hence, to decrease the number of data-flow dependencies and to increase the amount of complete summaries that can be pre-computed offline, it is advisable to make use of shortcuts whenever possible.

6 Limitations of the Approach

In this section, we briefly discuss the limitations of ModAlyzer. ModAlyzer needs to summarize the different pieces of information presented in Figure 3 to be able to construct effective module-wise summaries for a given concrete client analysis. Hence, ModAlyzer requires analysis algorithms that produce summarizable results such as IFDS [55], IDE [63] or *Weighted Pushdown Systems* (WPDS) [56].

For problems that are distributive, hence fit into these frameworks, the summarization is lossless. It is generally also possible to use ModAlyzer to solve non-distributive client analysis problems. As mentioned in Section 1, one cannot generally compute summaries for non-distributive data-flow problems. In that case, the approach can only make use of the summaries for type-hierarchy, points-to, and callgraph information, which may still lead to large performance increases as we present in Section 5.

We use never-invalidating points-to information computed using an *Andersen* [19] or *Steensgaard*-style [73] algorithm to be able to produce effective summaries. Again, computing more precise inter-procedural, context-, and flow-sensitive points-to information is a non-distributive problem for which no effective summaries can be computed. However, Späth et al. showed how flow- and context-sensitive pointer analysis can be decomposed into multiple analysis problems each of which, in turn, can be expressed within a distributive framework [72] – making the overall problem distributive. ModAlyzer's current points-to algorithm could therefore also be replaced by an adjusted version the distributive Bommerang approach proposed by Späth et al. The Boomerang approach – as is – operates in an on-demand manner and does not compute reusable summaries nor does it persist results. It is interesting to see the performance of ModAlyzer with an improved Boomerang-style points-to algorithm, that reuses summaries, presented in [72], but we consider it as future work.

As described in Section 5, ModAlyzer's overall effectiveness degrades with the number of updates that must be performed while merging summaries with the application code. Therefore, ModAlyzer's performance increase may not apply to programs that make excessive use of callbacks.

7 Related Work

Several previous approaches address, in part, the difficult problem of compositional static analysis [30, 31, 33, 36, 38, 52, 59–61, 78, 83]. However, existing techniques for compositional static analysis typically focus on data-flow or points-to analysis only. As advocated in this paper, a concrete compositional data-flow analysis client requires at the very least a combination of compositional callgraph, points-to and data-flow analysis.

Compositional data-flow techniques rely on the functional approach [66] allowing to solve distributive data-flow problems by using summary-based, inherently compositional frameworks such as IFDS [55], IDE [63], or WPDS [56]. Rountev et al. used IDE data-flow summaries to summarize large object-oriented libraries [62] and showed that a significant amount of time can be saved when using pre-computed summaries. The approach presented by Rountev et al., however, omits to tackle the challenging task of persisting general IDE summaries but rather discards the summaries at analysis shutdown. StubDroid [21] is a fully automated approach to generate precise library models for taint-analysis problems for the Android Framework, effectively preventing the re-analysis of the Android Framework for the analysis of different Android apps. Both Rountev's approach and StubDroid assume the existence of whole-program points-to and callgraph information.

Several works use partial points-to information in from of function-local summaries computed using context-free language (CFL-)reachability [48, 65, 81]. The summaries can be used in various scenarios allowing, among others, for on-demand points-to analysis, pre-analysis, and pointer analysis of partial programs using different sensitivities. These works present individual solutions to individual problems, while this paper presents the first integrated approach and shows its effectiveness on real-world C/C++ applications.

The IDEal [71] approach developed by Späth et al. is an alias-aware extension to the framework IDE [63] framework. IDEal embeds the alias analysis Boomerang [72] into the IDE solver implementation HEROS [25] to automatically resolve alias queries on-demand at analysis time while solving a given distributive data-flow analysis problem. However, it does not compute (persisted,) reusable summaries but rather computes analysis queries on-demand and still requires external callgraph graph information.

AVERROES [18] uses the *separate compilation assumption* and Java's constant pool [9] to generate sound and precise callgraphs without actually analyzing library code in order to generate a placeholder library. Existing whole-program callgraph construction algorithms can use the replacement to obtain a sound and precise application callgraph. AVERROES supports callgraph construction only. Its summaries cannot be used for precise pointer analysis, nor for precise data-flow analysis.

Other techniques try to improve the scalability of inter-procedural static analysis by using sparse propagation of data-flow facts along def-use chains [77] or demand-driven analysis that only analyze parts of a program that a user is currently interested in [72, 76]. Sparseness is a concept orthogonal to the ones proposed here. Both could be used in combination.

Some tools, including clang-tidy [2] and CppCheck [5], trade off scalability for reduced complexity. Thus, they only apply syntactic analysis to retrieve information on the property of interest. Precise, fully-fledged static analysis is replaced by much simpler checks that are capable of analyzing even million lines of code in minutes. However, these checks are often too imprecise to check for interesting properties.

Klohs et al. described the situation for *may*-analysis in which $\top$, representing all information, is obtained along one path in the control-flow graph, and thus, the other path does not have to be analyzed. This allows to remove data-flow dependencies ahead of time [43]. The approach presented here adopts this insight.

ModAlyzer computes the module-level summaries in a completely unrestricted way and does not make any assumptions about missing code. Yet, it may be advisable to compute summaries based on various sensible assumptions in scenarios where the summarization step can be performed ahead of time, e.g. for library pre-analysis. Tree-adjoining languages [79] and Dyck context-free language reachability [30, 78] can be used to increase the effective library summarization by computing reasonable conditional summaries that enable greater summary reuse under certain premises checked at analysis time of the application code. Such a strategy allows for more computations to be performed on a module-level. During the merge, the analysis can check whether an assumption that has been made holds and, if so, directly use the corresponding summary that may be much more expressive than one that has been computed without any assumptions about missing code, effectively reducing the amount of work that needs to be done while merging summaries with the application code. ModAlyzer currently does not use such a conditional summarization, however, it provides all required infrastructure to easily integrate the approach. Unfortunately, one cannot rely on programmers specifying pointer or reference parameters as constants using the `const` keyword because C/C++'s typesystem provides several mechanisms to circumvent constant declarations (e.g. `const_cast` and `mutable` in case of C++). Although writes through `const` are possible, they are used sparingly in real-world software as shown by Eyolfson and Lam [35]. Therefore, one reasonable assumption may be *const means const*. Especially `const`-qualified pointer parameters then represent hard inter-procedural boundaries and a data-flow analysis is not concerned with those parameters.

Early versions of Facebook's Infer [27] used separation logic to allow for the compositional analysis of heap-based programs. The approach computed bottom-up summaries using bi-abductive inference [24, 28], which could then be used in different calling contexts. Using Infer, one could thus formulate compositional static analyses that are evaluated using abstract interpretation. These analyses, however, were largely restricted to finding cases of memory corruption. Since about 2019 – reportedly due to a lack of general applicability and extensibility – Infer thus does not use abductive inference for most of its analyses any longer, and now instead bases its implementation on data-flow analysis using abstract interpretation. This analysis is no longer compositional.

8 Conclusion

In this paper, we presented ModAlyzer, a compositional approach to speeding up static analysis using persisted summaries for callgraph, points-to and data-flow information. We have presented an integrated strategy based on the dependencies as shown in Figure 3 that manages all those information and their dependencies, which many useful, concrete client analyses impose to provide precise results. ModAlyzer allows one to compute static analysis summaries on individual parts of a program without the need to make any assumptions on the missing code. These pre-computed summaries can then be (re)used later on, effectively shifting large parts of the computational effort to an offline phase.

Our experiments confirm the finding by previous works that actual application code often only constitutes only a small fraction of the complete program. Thus, ModAlyzer outperforms traditional whole program analysis in both runtime and flexibility.

References

1 C++ applications, December 2018. URL: `http://www.stroustrup.com/applications.html/`.
2 clang-tidy, August 2018. URL: `http://clang.llvm.org/extra/clang-tidy/`.

3 coreutils, July 2018. URL: https://www.gnu.org/software/coreutils/coreutils.html.

4 Coverity static application security testing (sast), December 2018. URL: https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html.

5 Cppcheck, August 2018. URL: http://cppcheck.sourceforge.net/.

6 Gcc optimize options, December 2018. URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

7 Grammatech codesonar, 2018. URL: https://www.grammatech.com/products/codesonar.

8 Intel® c++ compiler 19.0 developer guide and reference: Interprocedural optimization (ipo), December 2018. URL: https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-interprocedural-optimization-ipo.

9 Java virtual machine specification: The constant pool, December 2018. URL: https://docs.oracle.com/javase/specs/jvms/se10/html/jvms-4.html#jvms-4.4.

10 Phasar, July 2018. URL: https://phasar.org.

11 The programming languages beacon, December 2018. URL: http://www.lextrait.com/vincent/implementations.html/.

12 The state of open source security, December 2018. URL: https://snyk.io/stateofossecurity/.

13 Thinlto: Scalable and incremental lto, July 2018. URL: http://blog.llvm.org/2016/06/thinlto-scalable-and-incremental-lto.html.

14 Boost.serialization, August 2019. URL: https://www.boost.org/doc/libs/1_70_0/libs/serialization/doc/.

15 Graphviz, August 2019. URL: https://www.graphviz.org/.

16 Supplementary material, 2019. URL: https://drive.google.com/drive/folders/1uLHDkmdWdjQ-aeZjyRizhy9zwrX4VWVo?usp=sharing.

17 Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in c++ programs. In Pierre Cointe, editor, *ECOOP '96 — Object-Oriented Programming*, pages 142–166, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

18 Karim Ali and Ondřej Lhoták. Averroes: Whole-program analysis without the whole program. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 378–400, Berlin, Heidelberg, 2013. Springer-Verlag. doi:10.1007/978-3-642-39038-8_16.

19 Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. Datalogisk Institut, Københavns Universitet, 1994.

20 Steven Arzt and Eric Bodden. Reviser: Efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 288–298, New York, NY, USA, 2014. ACM. doi:10.1145/2568225.2568243.

21 Steven Arzt and Eric Bodden. Stubdroid: Automatic inference of precise data-flow summaries for the android framework. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 725–735, New York, NY, USA, 2016. ACM. doi:10.1145/2884781.2884816.

22 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM. doi:10.1145/2594291.2594299.

23 Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010. doi:10.1145/1646353.1646374.

24 Dirk Beyer, Sumit Gulwani, and David A Schmidt. Combining model checking and data-flow analysis. In *Handbook of Model Checking*, pages 493–540. Springer, 2018.

25 Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 3–8, New York, NY, USA, 2012. ACM. doi:10.1145/2259051.2259052.

26 Eric Bodden. The secret sauce in efficient and precise static analysis. In *Proceedings of the 7th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2018, 2018. To appear.

27 Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.

28 Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 289–300, New York, NY, USA, 2009. ACM. doi:10.1145/1480881.1480917.

29 Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in c++ programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 397–408, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/174675.177973.

30 Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. Optimal dyck reachability for data-dependence and alias analysis. *Proc. ACM Program. Lang.*, 2(POPL):30:1–30:30, 2017. doi:10.1145/3158118.

31 Michael Codish, Saumya K. Debray, and Roberto Giacobazzi. Compositional analysis of modular logic programs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 451–464, New York, NY, USA, 1993. ACM. doi:10.1145/158511.158703.

32 Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in c++. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, page 306–323, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/236337.236369.

33 M. B. Dwyer. Modular flow analysis for concurrent software. In *Proceedings of the 12th International Conference on Automated Software Engineering (Formerly: KBSE)*, ASE '97, pages 264–, Washington, DC, USA, 1997. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=786767.786816.

34 Michael Eichberg, Ben Hermann, Mira Mezini, and Leonid Glanz. Hidden truths in dead software paths. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 474–484, New York, NY, USA, 2015. ACM. doi:10.1145/2786805.2786865.

35 Jon Eyolfson and Patrick Lam. C++ const and Immutability: An Empirical Study of Writes-Through-const. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:25, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2016.8.

36 Sanjay Ghemawat, Keith H. Randall, and Daniel J. Scales. Field analysis: Getting useful and low-cost interprocedural information. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 334–344, New York, NY, USA, 2000. ACM. doi:10.1145/349299.349343.

37 Mark Harman and Peter O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23. IEEE, 2018.

38 Mary Jean Harrold and Gregg Rothermel. Separate computation of alias information for reuse. *IEEE Trans. Softw. Eng.*, 22(7):442–460, July 1996. doi:10.1109/32.538603.

39 Ben Hermann, Michael Reif, Michael Eichberg, and Mira Mezini. Getting to know you: Towards a capability model for java. In *Proceedings of the 2015 10th Joint Meeting on*

Foundations of Software Engineering, ESEC/FSE 2015, pages 758–769, New York, NY, USA, 2015. ACM. doi:10.1145/2786805.2786829.

40 P. Holzinger, B. Hermann, J. Lerch, E. Bodden, and M. Mezini. Hardening java's access control by abolishing implicit privilege elevation. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1027–1040, May 2017. doi:10.1109/SP.2017.16.

41 John B Kam and Jeffrey D Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

42 Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM. doi:10.1145/512927.512945.

43 Karsten Klohs. A summary function model for the validation of interprocedural analysis results. In *Proceedings of the 7th International Workshop on Compiler Optimization meets Compiler Verification, COCV'08*, 2008.

44 Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. Cognicrypt: Supporting developers in using cryptography. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 931–936, Piscataway, NJ, USA, 2017. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=3155562.3155681.

45 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

46 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 129–140, New York, NY, USA, 2018. ACM. doi:10.1145/3236024.3236041.

47 V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1251398.1251416.

48 Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An incremental points-to analysis with cfl-reachability. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, pages 61–81, Berlin, Heidelberg, 2013. Springer-Verlag. doi:10.1007/978-3-642-37051-9_4.

49 Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005.

50 Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, Inc., 1st edition, 2014.

51 Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical extensions to the ifds algorithm. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 124–144, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.1007/978-3-642-11970-5_8.

52 Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In Ole Lehrmann Madsen, editor, *ECOOP '92 European Conference on Object-Oriented Programming*, pages 329–349, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

53 Santanu Saha Ray. *Graph Theory with Algorithms and Its Applications: In Applied Science and Technology*. Springer Publishing Company, Incorporated, 2014.

54 Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for java libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International*

Symposium on Foundations of Software Engineering, FSE 2016, pages 474–486, New York, NY, USA, 2016. ACM. doi:10.1145/2950290.2950312.

55 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM. doi:10.1145/199448.199462.

56 Thomas Reps, Stefan Schwoon, and Somesh Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 189–213, Berlin, Heidelberg, 2003. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1760267.1760283.

57 H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society, 1953, 74, 2, 358*, 1953.

58 Personal communication with atanas (nasko) rountev, 2014.

59 A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in java software. *IEEE Transactions on Software Engineering*, 30(6):372–387, June 2004. doi:10.1109/TSE.2004.20.

60 Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In Reinhard Wilhelm, editor, *Compiler Construction*, pages 20–36, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

61 Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering — ESEC/FSE '99*, pages 235–252, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

62 Atanas Rountev, Mariana Sharp, and Guoqing Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 53–68, Berlin, Heidelberg, 2008. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1788374.1788380.

63 Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1-2):131–170, 1996. doi:10.1016/0304-3975(96)00072-2.

64 Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Phasar: An inter-procedural static analysis framework for c/c++. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 393–410, Cham, 2019. Springer International Publishing.

65 Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 264–274, New York, NY, USA, 2012. ACM. doi:10.1145/2259016.2259050.

66 M Sharir and A Pnueli. *Two approaches to interprocedural data flow analysis*. New York Univ. Comput. Sci. Dept., New York, NY, 1978. URL: https://cds.cern.ch/record/120118.

67 Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library - User Guide and Reference Manual*. C++ in-depth series. Pearson / Prentice Hall, 2002.

68 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, New York, NY, USA, 2011. ACM. doi:10.1145/1926385.1926390.

69 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, New York, NY, USA, 2014. ACM. doi:10.1145/2594291.2594320.

70 Black Duck Software. 2018 open source security and risk analysis. https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2018-ossra.pdf, 2018.

71 Johannes Späth, Karim Ali, and Eric Bodden. Ideal: Efficient and precise alias-aware dataflow analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi:10.1145/3133923.

72 Johannes Späth, Lisa Nguyen, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In *European Conference on Object-Oriented Programming (ECOOP)*, 17 - 22 July 2016.

73 Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM. doi:10.1145/237721.237727.

74 R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986. doi:10.1109/TSE.1986.6312929.

75 Robert E. Strom. Mechanisms for compile-time enforcement of security. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 276–284, New York, NY, USA, 1983. ACM. doi:10.1145/567067.567093.

76 Yulei Sui and Jingling Xue. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 460–473, New York, NY, USA, 2016. ACM. doi:10.1145/2950290.2950296.

77 Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 265–266, New York, NY, USA, 2016. ACM. doi:10.1145/2892208.2892235.

78 Hao Tang, Di Wang, Yingfei Xiong, Lingming Zhang, Xiaoyin Wang, and Lu Zhang. Conditional dyck-cfl reachability analysis for complete and efficient library summarization. In Hongseok Yang, editor, *Programming Languages and Systems*, pages 880–908, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

79 Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 83–95, New York, NY, USA, 2015. ACM. doi:10.1145/2676726.2676997.

80 John Toman and Dan Grossman. Taming the Static Analysis Beast. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.SNAPL.2017.18.

81 John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, page 187–206, New York, NY, USA, 1999. Association for Computing Machinery. doi:10.1145/320384.320400.

82 Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. Shape analysis. In *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, pages 1–17, London, UK, UK, 2000. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=647476.760384.

83 Jingling Xue and Phung Hua Nguyen. Completeness analysis for incomplete object-oriented programs. In *International Conference on Compiler Construction*, volume 3443, pages 271–286, April 2005. doi:10.1007/978-3-540-31985-6_21.

Gradual Program Analysis for Null Pointers

Sam Estep
Carnegie Mellon University, Pittsburgh, PA, USA

Jenna Wise
Carnegie Mellon University, Pittsburgh, PA, USA

Jonathan Aldrich
Carnegie Mellon University, Pittsburgh, PA, USA

Éric Tanter
Computer Science Department (DCC), University of Chile, Santiago, Chile

Johannes Bader
Jane Street, New York, NY, USA

Joshua Sunshine
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Static analysis tools typically address the problem of excessive false positives by requiring programmers to explicitly annotate their code. However, when faced with incomplete annotations, many analysis tools are either too conservative, yielding false positives, or too optimistic, resulting in unsound analysis results. In order to flexibly and soundly deal with partially-annotated programs, we propose to build upon and adapt the gradual typing approach to abstract-interpretation-based program analyses. Specifically, we focus on null-pointer analysis and demonstrate that a gradual null-pointer analysis hits a sweet spot, by gracefully applying static analysis where possible and relying on dynamic checks where necessary for soundness. In addition to formalizing a gradual null-pointer analysis for a core imperative language, we build a prototype using the Infer static analysis framework, and present preliminary evidence that the gradual null-pointer analysis reduces false positives compared to two existing null-pointer checkers for Infer. Further, we discuss ways in which the gradualization approach used to derive the gradual analysis from its static counterpart can be extended to support more domains. This work thus provides a basis for future analysis tools that can smoothly navigate the tradeoff between human effort and run-time overhead to reduce the number of reported false positives.

2012 ACM Subject Classification Software and its engineering → General programming languages; Software and its engineering → Software verification

Keywords and phrases gradual typing, gradual verification, dataflow analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.3

Related Version *Full Version*: `https://arxiv.org/abs/2105.06081` [10]

Supplementary Material *Software*: `https://github.com/orgs/gradual-verification/packages/container/package/ecoop21`

Funding National Science Foundation under Grant No. CCF-1901033 and Grant No. CCF-1852260
Éric Tanter: FONDECYT Regular project 1190058

1 Introduction

Static analysis is useful [1], but underused in practice because of false positives [15]. A commonly-used way to reduce false positives is through programmer-provided annotations [4] that make programmers intent manifest. For example, Facebook's Infer Eradicate [11], Uber's NullAway [3], and the Java Nullness Checker from the Checker Framework [21] all rely on `@NonNull` and `@Nullable` annotations to statically find and report potential null-pointer

exceptions in Java code. However, in practice, annotating code completely can be very costly [6] – or even impossible, for instance, when relying on third-party libraries and APIs. As a result, since non-null reference variables are used extensively in software [6], many tools assume missing annotations are `@NonNull`. But, the huge number of false positives produced by such an approach in practice is a serious burden. To address this pitfall, NullAway assumes that sinks (i.e. targets of assignments and bindings) are `@Nullable` and sources are `@NonNull`. Unfortunately, both strategies are unsound, and therefore programs deemed valid may still raise null pointer exceptions at run time.

This paper explores a novel approach to these issues by drawing on research in gradual typing [22, 23, 14] and its recent adaptation to gradual verification [2, 24]. We propose gradual program analysis as a principled, sound, and practical way to handle missing annotations. As a first step in the research agenda of gradual program analysis, this article studies the case of a simple null-pointer analysis. We present a general formal framework to derive gradual program analyses by transforming static analyses based on abstract interpretation [8]. Specifically, we study analyses that operate over first-order procedural imperative languages and support user-provided annotations. This setting matches the core language used by many tools, such as Infer. In essence, a *gradual analysis* treats missing annotations optimistically, but injects run-time checks to preserve soundness. Crucially, the static portion of a gradual analysis uses the same algorithmic architecture as the underlying static analysis.[1]

Additionally, we ensure that any gradual analysis produced from our framework satisfies the *gradual guarantees*, adapted from Siek *et al.* [23] formulation for gradual typing. Any gradual analysis is also a *conservative extension* of the base static analysis: when all annotations are provided, the gradual analysis is equivalent to the base static analysis, and no run-time checks are inserted. Therefore, the gradual analysis smoothly trades off between static and dynamic checking, driven by the annotation effort developers are willing to invest.

To provide initial evidence of the applicability of gradual null-pointer analysis, we implement a gradual null-pointer analysis (GNPA) using Facebook's Infer analysis framework and report on preliminary experiments using the prototype.[2] The experiments show that a gradual null-pointer analysis can be effectively implemented, and used at scale to produce a small number of false positives in practice – fewer than Infer Eradicate as well as a more recent Infer checker, NullSafe. They also show that GNPA eliminates on average more than half of the null-pointer checks Java automatically inserts at run time. As a result, unlike other null-pointer analyses, GNPA can both prove the redundancy of run-time checks and reduce reported false positives.

The rest of the paper is organized as follows. In Section 2, we motivate gradual program analysis in the setting of null pointers by looking at how Eradicate, NullSafe, NullAway, and the Java Nullness Checker operate on example code with missing annotations, showcasing the concrete advantages of GNPA. Section 3 formalizes PICL, a core imperative language similar to that of Infer. Section 4 then presents the static null-pointer analysis (NPA) for PICL, which is then used as the starting point for the derivation of the gradual analysis. We describe our approach to gradualizing a static program analysis in Section 5, using GNPA as the running case study. Additionally, Section 5 includes a discussion of important gradual properties our analysis adheres to: *soundness*, *conservative extension*, and the *gradual guarantee*. All proofs can be found in the appendix of the full version of this paper [10]. We

[1] Note that an alternative is phrasing nullness as a type system, which can also be gradualized [5, 19]. We focus on approaches based on static analysis, which have very different technical foundations and user experience. We compare to type-based approaches in Section 7.

[2] `https://github.com/orgs/gradual-verification/packages/container/package/ecoop21`

report on the preliminary empirical evaluation of an Infer GNPA checker called *Graduator* in Section 6. Section 7 discusses related work and Section 8 concludes. In the conclusion, we sketch ways in which the approach presented here could be applied to other analysis domains, highlight open venues for future work in the area of gradual program analysis.

2 Gradual Null-Pointer Analysis in Action

This section informally introduces gradual null-pointer analysis and its potential compared to existing approaches through a simple example. We first briefly recall the basics of null-pointer analyses, and then discuss how current tools deal with missing annotations in problematic ways.

2.1 Null-Pointer Analysis in a Nutshell

With programming languages that allow any type to be inhabited by a null value, programmers end up facing runtime errors (or worse if the language is unsafe) related to dereferencing null pointers. A null-pointer analysis is a static analysis that detects *potential* null pointer dereferences and reports them as warnings, so that programmers can understand where explicit nullness checks should be added in order to avoid runtime errors. Examples of null-pointer analyses are Infer Eradicate [12] and the Java Null Checker [21]. Typically, a null-pointer analysis allows programmers to add annotations in the code to denote which variables (as well as fields, return values, etc.) are, or should be, non-null–e.g. `@NonNull`–and which are potentially null–e.g. `@Nullable`. A simple flow analysis is then able to detect and report conflicts, such as when a nullable variable is assigned to a non-null variable.

While a static null pointer analysis brings guarantees of robustness to a codebase, its adoption is not necessarily seamless. If a static analysis aims to be sound, it must not suffer from false negatives, i.e. miss any actual null pointer dereference that can happen at runtime. While desirable, this means the analysis necessarily has to be conservative and therefore reports false positives – locations that are thought to potentially trigger null pointer dereferences, but actually do not.

This standard static analysis conundrum is exacerbated when considering programs where not all variables are annotated. Of course, in practice, a codebase is rarely fully annotated. Existing null-pointer analyses assign missing annotations a concrete annotation, such as `Nullable` or `NonNull`. In doing so, they either report additional false positives, suffer from false negatives (and hence are unsound), or both. The rest of this section illustrates these issues with a simple example, and discusses how a gradual null-pointer analysis (GNPA) alleviates them. GNPA treats missing annotations in a special manner, following the gradual typing motto of being optimistic statically and relying on runtime checks for soundness [22]. Doing so allows the analysis to leverage both static and dynamic information to reduce false positives while maintaining soundness.

2.2 Avoiding False Positives

GNPA can reduce the number of false positives reported by static tools by leveraging provided annotations and run-time checks. We demonstrate this with the unannotated program in Figure 1. The program appends the reverse of a non-null string to the reverse of a null string and prints the result. The `reverse` method (lines 3–8) returns the reverse of an input string when it is non-null and an empty string when the input is `null`. Additionally, `reverse` is unannotated, as highlighted for reference.

```
1  class Main {
2
3    static    String reverse(    String str) {
4      if (str == null) return new String();
5      StringBuilder builder = new StringBuilder(str);
6      builder.reverse();
7      return builder.toString();
8    }
9
10   public static void main(String[] args) {
11     String reversed = reverse(null);
12     String frown = reverse(":)");
13     String both = reversed.concat(frown);
14     System.out.println(both);
15   }
16 }
```

Figure 1 Unannotated Java code safely reversing nullable strings.

The most straightforward approach to handling the missing annotations is to replace them with a fixed annotation. Infer Eradicate and the Java Nullness Checker both choose `@NonNull` as the default, since that is the most frequent annotation used in practice [6]. Thus, in this example, they would treat `reverse`'s argument and return value as annotated with `@NonNull`. This correctly assigns `reversed` and `frown` as non-null on lines 11 and 12; and consequently, no false positive is reported when `reversed` is dereferenced on line 13. However, both tools will report a false positive each time `reverse` is called with `null`, as in line 11.

Other uniform defaults are possible, but likewise lead to false positive warnings. For example, choosing `@Nullable` by default would result in a false positive when `reversed` is dereferenced. A more sophisticated choice would be the Java Nullness Checker's `@PolyNull` annotation, which supports type qualifier polymorphism for methods annotated with `@PolyNull`. If `reverse`'s method signature is annotated with `@PolyNull`, then `reverse` would have two conceptual versions:

```
static @Nullable String reverse(@Nullable String str)
 static @NonNull String reverse(@NonNull String str)
```

At a call site, the most precise applicable signature would be chosen; so, calling `reverse` with `null` (line 11) would result in the `@Nullable` signature, and calling `reverse` with `":)"` (line 12) would result in the `@NonNull` signature. Unfortunately, this strategy marks `reversed` on line 11 as `@Nullable` even though it is `@NonNull`, and a false positive is reported when `reversed` is dereferenced on line 13. So while `@PolyNull` increases the expressiveness of the annotation system, it does not solve the problem of avoiding false positives from uniform annotation defaults.

In contrast, GNPA optimistically assumes both calls to `reverse` in `main` (lines 11–12) are valid without assigning fixed annotations to `reverse`'s argument or return value. Then, the analysis can continue relying on *contextual optimism* when reasoning about the rest of `main`: `reversed` is assumed `@NonNull` to satisfy its dereference on line 13. Of course this is generally an unsound assumption, so a run-time check is inserted to ascertain the

non-nullness of `reversed` and preserve soundness. Alternatively, a developer could annotate the return value of `reverse` with `@NonNull`. GNPA will operate as before except it will leverage this new information during static reasoning. Therefore, `reversed` will be marked `@NonNull` on line 11 and the dereference of `reversed` on line 13 will be statically proven safe without any run-time check.

It turns out that a non-uniform choice of defaults can be optimistic in the same sense as GNPA. For example, NullAway assumes sinks are `@Nullable` and sources are `@NonNull` when annotations are missing. In fact, this strategy correctly annotates `reverse`, and so no false positives are reported by the tool for the program in Figure 1. However, in contrast to the gradual approach, the NullAway approach is in fact unsound, as illustrated next.

2.3 Avoiding False Negatives

When Eradicate, NullAway, and the Java Nullness Checker handle missing annotations, they all give up soundness in an attempt to limit the number of false positives produced.

To illustrate, consider the same program from Figure 1, with one single change: the `reverse` method now returns `null` instead of an empty string (line 4).

```
if (str == null) return null;
```

All of the tools mentioned earlier, including NullAway, erroneously assume that the return value of `reverse` is `@NonNull`. On line 11, `reversed` is assigned `reverse(null)`'s return value of `null`; so, it is an error to dereference `reversed` on line 13. Unfortunately, all of the tools assume `reversed` is assigned a non-null value and do not report an error on line 13. This is a *false negative*, which means that at runtime the program will fail with a null-pointer exception.

GNPA is similarly optimistic about `reversed` being non-null on line 13. However, GNPA safeguards its optimistic static assumptions with run-time checks. Therefore, the analysis will correctly report an error on line 13. Alternatively, a developer could annotate the return value of `reverse` with `@Nullable`. By doing so, the gradual analysis will be able to exploit this information statically to report a static error, instead of a dynamic error.

To sum up, a gradual null-pointer analysis can reduce false positives by optimistically treating missing annotations, and preserve soundness by detecting errors at runtime. Of course, one may wonder why it is better to fail at runtime when passing a null value as a non-null annotated argument, instead of just relying on the upcoming null-pointer exception. There are two answers to this question. First, in unsafe languages like C, a null-pointer dereference results in a crash. Second, in a safe language like Java where a null-pointer dereference is anyway detected and reported, it can be preferable to fail as soon as possible, in order to avoid performing computation (and side effects) under an incorrect assumption. This is similar to how the eager reporting of gradual typing can be seen as an improvement over simply relying on the underlying safety checks of a dynamically-typed language.

Next, we formally develop GNPA, prove that it is sound, and prove that it smoothly trades-off between static and dynamic checking following the gradual guarantee criterion from gradual typing [23]. We finally report on an actual implementation of GNPA and compare its effectiveness with existing tools.

$$
\begin{array}{rcl}
x, y & \in & \textsc{Var} \\
e & \in & \textsc{Expr} \\
a & \in & \textsc{Ann} = \{\texttt{Nullable}, \texttt{NonNull}, \texttt{?}\} \\
P & ::= & \overline{\mathit{procedure}}\ \overline{\mathit{field}}\ s \\
\mathit{field} & ::= & T\ f; \\
\mathit{procedure} & ::= & T\texttt{@}a\ m\ (\ \overline{T\texttt{@}a\ x}\ )\ \{\ s\ \} \\
T & ::= & \texttt{ref} \\
\oplus & ::= & \wedge \mid \vee
\end{array}
\qquad
\begin{array}{rcl}
m & \in & \textsc{Proc} \\
f & \in & \textsc{Field} \\
s & \in & \textsc{Stmt} \\
e & ::= & \texttt{null} \mid x \mid e \oplus e \mid e.f \mid \texttt{new}(\overline{f}) \\
 & & \mid m(x) \\
c & ::= & e = \texttt{null} \mid e \neq \texttt{null} \\
s & ::= & \texttt{skip} \mid s\ ;\ s \mid T\ x \mid x\ \texttt{:=}\ e \\
 & & \mid x.f\ \texttt{:=}\ y \mid \texttt{if}\ (c)\ \{\ s\ \}\ \texttt{else}\ \{\ s\ \} \\
 & & \mid \texttt{while}\ (c)\ \{\ s\ \} \mid \texttt{return}\ y
\end{array}
$$

Figure 2 Abstract syntax of PICL.

3 PICL: A Procedural Imperative Core Language

Following the Abstract Gradual Typing methodology introduced by Garcia *et al.* [14], we build GNPA on top of a static null-pointer analysis, NPA. Thus, we first formally present a procedural imperative core language (PICL), used for both analyses to operate on; we present NPA in Section 4, and GNPA in Section 5. PICL is akin to the intermediate language of the Infer framework, and therefore the formal development around PICL drove the implementation of the Infer GNPA checker we evaluate in Section 6.

3.1 Syntax & Static Semantics

The syntax of PICL can be found in Figure 2. Programs consist of procedures[3], fields, and statements. Statements include the empty statement, sequences, variable declarations, variable and field assignments, conditionals, while loops, and returns. Expressions consist of `null` literals, variables, comparisons, conjunctions, disjunctions, field accesses, object allocations, and procedure calls. Finally, procedures may have `Nullable` or `NonNull` annotations on their arguments and return values. Missing annotations are represented by `?`.

As the focus of this work is not on typing, we only consider well-formed and well-typed programs, which is standard and not formalized here. In particular, variables are declared and initialized before use, and field and procedure names are unique.

3.2 Control Flow Graph Representation

Well-formed programs written in the abstract syntax given in Fig. 2 are translated into *control flow graphs* – one graph for each procedure body and one for the main s. A finite control flow graph (CFG) for program p has vertices $\textsc{Vert}_p$ and edges $\textsc{Edge}_p \subseteq \textsc{Vert}_p \times \textsc{Vert}_p$. For $v_1, v_2 \in \textsc{Vert}_p$, we write $v_1 \xrightarrow{p} v_2$ to denote $(v_1, v_2) \in \textsc{Edge}_p$. Each vertex holds a single instruction, which we can access using the function $\textsc{inst}_p : \textsc{Vert}_p \to \textsc{Inst}$. We write $[\iota]_v$ to denote a vertex $v \in \textsc{Vert}_p$ such that $\textsc{inst}_p(v) = \iota$, or just $[\iota]$ (omitting the v) when the vertex itself is not important. By construction, these translated CFGs satisfy certain well-formedness properties, listed in the appendix of the full version of this paper [10].

The set of possible instructions is defined in Figure 3. In general, the CFG instructions are atomic variants of program statements designed to simplify the analysis presentations. Figure 4 gives the CFG of a simple procedure `foo`, which calls `bar` repeatedly until `x` becomes non-null

[3] Procedures accept only one parameter to simplify later formalisms.

$x, y, z \in \textsc{Var}$ $\quad m \in \textsc{Proc}$

$a, b \in \textsc{Ann} = \{\texttt{Nullable}, \texttt{NonNull}, \texttt{?}\}$ $\quad f \in \textsc{Field}$

$$I ::= x \texttt{ := } y \mid x \texttt{ := null} \mid x \texttt{ := } m\texttt{@}a(y\texttt{@}b) \mid x \texttt{ := new}(\overline{f}) \mid x \texttt{ := } y \wedge z \mid x \texttt{ := } y \vee z$$
$$\mid x \texttt{ := } y.f \mid x.f \texttt{ := } y \mid \texttt{branch } x \mid \texttt{if } x \mid \texttt{else } x \mid \texttt{return } y\texttt{@}a \mid \texttt{main}$$
$$\mid \texttt{proc } m\texttt{@}a(y\texttt{@}b)$$

Figure 3 Abstract syntax of a CFG instruction.

```
1  ref@NonNull foo(ref@Nullable x)
2  {
3    while (x == null)
4    {
5      x := bar(x);
6    }
7    return x;
8  }
```

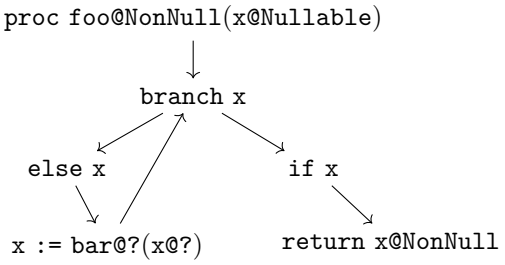


Figure 4 Example CFG.

and then returns `x`. The CFG starts with `foo`'s entry node `proc` $foo@NonNull(x@Nullable)$ (similarly, `main` is always the entry node of the main program's CFG). Then, the while loop on lines 3–6 results in the `branch` x sub-graph, which leads to `if` x when x is non-null and `else` x when x is null. The call to `bar` follows from `else` x and loops back to `branch` x as expected. Finally, `return` $x@NonNull$ follows from `if` x ending the CFG. Precise semantics for instructions is given in Section 3.3.

3.3 Dynamic Semantics

We define the set of possible object locations as the set of natural numbers and 0, $\textsc{Val} = \mathbb{N} \cup \{0\}$. The `null` pointer is location 0.

Now, a program state ($\textsc{State}_p \subseteq \textsc{Stack}_p \times \textsc{Mem}_p$) consists of a stack and a heap. A heap $\mu \in \textsc{Mem}_p = (\textsc{Val} \setminus \{0\}) \rightharpoonup (\textsc{Field} \rightharpoonup \textsc{Val})$ maps object locations and field names to program values – other (possibly null) pointers. A stack is made of stack frames each containing a local variable environment and CFG node:

$$S \in \textsc{Stack}_p ::= E \cdot S \mid \mathsf{nil} \quad \text{where} \quad E \in \textsc{Frame}_p = \textsc{Env} \times \textsc{Vert}_p$$
$$\text{and} \quad \textsc{Env} = \textsc{Var} \rightharpoonup \textsc{Val}.$$

Further, we restrict the set of states $\xi = \langle\langle \rho_1, v_1\rangle \cdot \langle \rho_2, v_2\rangle \cdots \langle \rho_n, v_n\rangle \cdot \mathsf{nil} \parallel \mu\rangle \in \textsc{State}_p$ to include only those satisfying the following conditions:

1. *Bottom stack frame is in* ***`main`****:* Let $\textsc{descend} : \textsc{Vert}_p \to \mathcal{P}^+(\textsc{Vert}_p)$ give the descendants of each node in the control flow graph. Then $v_i \in \textsc{descend}(v_0)$ if and only if $i = n$.
2. *Every variable defaults to* ***`null`*** *(except on* ***`main`*** *and* ***`proc`*** *nodes):* If $\textsc{inst}_p(v_i) \neq \texttt{main}$ and $\textsc{inst}_p(v_i) \neq \texttt{proc } m@a(y@b)$ then ρ_i is a total function.
3. *Follow the "true" branch when non-null:* If $\textsc{inst}_p(v_i) = \texttt{if } y$ then $\rho_i(y) \neq 0$.
4. *Follow the "false" branch when null:* If $\textsc{inst}_p(v_i) = \texttt{else } y$ then $\rho_i(y) = 0$.

$$\langle\langle \rho, [x \texttt{ := } y]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho[x \mapsto \rho(y)], v\rangle \cdot S \parallel \mu\rangle$$
$$\langle\langle \rho, [\texttt{branch } y]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho, [\textsc{branch}(\rho(y), y)]_v\rangle \cdot S \parallel \mu\rangle$$
$$\langle\langle \rho, [\texttt{if } y]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho, v\rangle \cdot S \parallel \mu\rangle$$
$$\langle\langle \rho, [\texttt{else } y]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho, v\rangle \cdot S \parallel \mu\rangle$$
$$\langle\langle \rho, [x \texttt{ := } m\texttt{@}a(y\texttt{@}b)]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \varnothing, [\texttt{proc } m\texttt{@}a(y'\texttt{@}b)]\rangle \cdot \langle \rho, u\rangle \cdot S \parallel \mu\rangle$$
$$\langle\langle \rho_1, [\texttt{proc } m\texttt{@}a(y\texttt{@}b)]_u\rangle \cdot \langle \rho_2, [x \texttt{ := } m\texttt{@}a(y'\texttt{@}b)]_w\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho_0[y \mapsto \rho_2(y')], v\rangle \cdot \langle \rho_2, w\rangle \cdot S \parallel \mu\rangle$$
$$\langle\langle \rho_1, [\texttt{return } y\texttt{@}a]\rangle \cdot \langle \rho_2, [x \texttt{ := } m\texttt{@}a(y'\texttt{@}b)]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho_2[x \mapsto \rho_1(y)], v\rangle \cdot S \parallel \mu\rangle \dagger$$
$$\langle\langle \rho, [x \texttt{ := null}]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho[x \mapsto 0], v\rangle \cdot S \parallel \mu\rangle$$
$$\langle\langle \rho, [x \texttt{ := new}(\overline{f})]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho[x \mapsto \textsc{new}(\mu)], v\rangle \cdot S \parallel \mu[\textsc{new}(\mu) \mapsto \overline{[f_i \mapsto \texttt{null}]}]\rangle$$
$$\langle\langle \rho, [x \texttt{ := } y \wedge z]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho[x \mapsto \textsc{and}(\rho(y), \rho(z))], v\rangle \cdot S \parallel \mu\rangle$$
$$\langle\langle \rho, [x \texttt{ := } y \vee z]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho[x \mapsto \textsc{or}(\rho(y), \rho(z))], v\rangle \cdot S \parallel \mu\rangle$$
$$\langle\langle \rho, [x \texttt{ := } y.f]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho[x \mapsto \mu(\rho(y))(f)], v\rangle \cdot S \parallel \mu\rangle$$
$$\langle\langle \rho, [x.f \texttt{ := } y]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho, v\rangle \cdot S \parallel \mu[\rho(x) \mapsto [f \mapsto \rho(y)]]\rangle$$
$$\langle\langle \rho, [\texttt{main}]_u\rangle \cdot S \parallel \mu\rangle \longrightarrow_p \langle\langle \rho_0, v\rangle \cdot S \parallel \mu\rangle$$

Figure 5 Small-step semantics rules that hold when $u \xrightarrow{p} v$. † This particular rule only applies if either $a = \texttt{?}$ or $\rho_1(y) \in \textsc{conc}(a)$ (see Section 4).

5. *Every frame except the top is a procedure call:* If $v_i \in \textsc{descend}(\texttt{proc } m\texttt{@}a(y\texttt{@}b))$ then $\textsc{inst}_p(v_{i+1}) = x \texttt{ := } m\texttt{@}a(y'\texttt{@}b)$, and either $b = \texttt{?}$ or $\rho_{i+1}(y') \in \textsc{conc}(b)$ (see section 4.

Now, the small-step semantics of PICL is given in Figure 5, where $\rho_0 = \{x \mapsto 0 : x \in \textsc{Var}\}$. The rules rely on the following helper functions:

$$\textsc{new} : \textsc{Mem}_p \to \textsc{Val} \setminus \{0\} \qquad \textsc{new}(\mu) = 1 + \max(\{0\} \cup \mathrm{dom}(\mu))$$
$$\textsc{branch} : \textsc{Val} \times \textsc{Var} \to \textsc{Inst} \qquad \textsc{branch}(n, x) = \texttt{if } x \text{ if } n > 0; \texttt{else } x \text{ otherwise}$$
$$\textsc{and} : \textsc{Val} \times \textsc{Val} \to \textsc{Val} \qquad \textsc{and}(n_1, n_2) = n_2 \text{ if } n_1 > 0; n_1 \text{ otherwise}$$
$$\textsc{or} : \textsc{Val} \times \textsc{Val} \to \textsc{Val} \qquad \textsc{or}(n_1, n_2) = n_1 \text{ if } n_1 > 0; n_2 \text{ otherwise}$$

Notably, `branch` y steps to the `if` y node when y is non-null and `else` y when y is null. Additionally, if a procedure call's argument disagrees with its parameter annotation, then it will get stuck (rule 5 for states); otherwise, the call statement will safely step to the procedure's body. In contrast, the semantics will get stuck if a return value does not agree with the procedure's return annotation.

4 A Static Null-Pointer Analysis for PICL

In this section, we formalize a static null-pointer analysis, called NPA, for PICL on which we will build GNPA. Here, we will only consider completely annotated programs, $\textsc{Ann} = \{\texttt{Nullable}, \texttt{NonNull}\}$. Therefore, we use a "prime" symbol for sets like $\textsc{Inst}' \subseteq \textsc{Inst}$ to indicate that this is not the whole story. We present NPA's semilattice of abstract values, flow function, fixpoint algorithm, and how the analysis uses the results from the fixpoint algorithm to report warnings to the user.

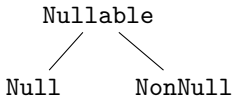


Figure 6 The ABST semilattice.

4.1 Semilattice of Abstract Values

The set of abstract values $\textsc{Abst} = \{\texttt{Nullable}, \texttt{Null}, \texttt{NonNull}\}$ make up the finite semilattice defined in Figure 6. The partial order $\sqsubseteq \subseteq \textsc{Abst} \times \textsc{Abst}$ given is

$$\texttt{Null} \sqsubseteq \texttt{Nullable} \qquad \texttt{NonNull} \sqsubseteq \texttt{Nullable} \qquad \forall.\ l \in \textsc{Abst}\ .\ l \sqsubseteq l.$$

The join function $\sqcup : \textsc{Abst} \times \textsc{Abst} \to \textsc{Abst}$ induced by the partial order is:

$$\texttt{Null} \sqcup \texttt{NonNull} = \texttt{Nullable} \qquad \forall.\ l \in \textsc{Abst}\ .\ l \sqcup \texttt{Nullable} = \texttt{Nullable}$$
$$\forall.\ l \in \textsc{Abst}\ .\ l \sqcup l = l$$

Clearly, `Nullable` is the top element $\top$. Next, we relate this semilattice to VAL via a concretization function $\textsc{conc} : \textsc{Abst} \to \mathcal{P}^+(\textsc{Val})$:

$$\textsc{conc}(\texttt{Nullable}) = \textsc{Val}, \quad \textsc{conc}(\texttt{Null}) = \{0\}, \quad \textsc{conc}(\texttt{NonNull}) = \textsc{Val} \setminus \{0\},$$

which satisfies the property $\forall.\ l_1, l_2 \in \textsc{Abst}\ .\ l_1 \sqsubseteq l_2 \iff \textsc{conc}(l_1) \subseteq \textsc{conc}(l_2)$.

4.2 Flow Function

Similar to how we use ENV to represent mappings from variables to concrete values, we will use $\sigma \in \textsc{Map} = \textsc{Var} \rightharpoonup \textsc{Abst}$ to represent mappings from variables to abstract values – *abstract states*. Then, we extend the semilattice's partial order relation to abstract states $\sigma_1, \sigma_2 \in \textsc{Map}$:

$$\sigma_1 \sqsubseteq \sigma_2 \iff \forall.\ x \in \textsc{Var}\ .\ \sigma_1(x) \sqsubseteq \sigma_2(x)$$

We also extend the join operation to abstract states $\sigma_1, \sigma_2 \in \textsc{Map}$:

$$(\sigma_1 \sqcup \sigma_2)(x) = \begin{cases} a \sqcup b & \text{if } \sigma_1(x) = a \text{ and } \sigma_2(x) = b \\ a & \text{if } \sigma_1(x) = a \text{ and } \sigma_2(x) \text{ is undefined} \\ b & \text{if } \sigma_1(x) \text{ is undefined and } \sigma_2(x) = b \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The NPA's flow function $\textsc{flow} : \textsc{Inst}' \times \textsc{Map} \to \textsc{Map}$ is defined in Figure 7. Note, $\sigma_0 = \{x \mapsto \texttt{Null} : x \in \textsc{Var}\}$. Also, we omit the `return` y`@`a case because it does not have CFG successors in a well-formed program.

4.2.1 Properties

It can be shown that this flow function is monotonic: for any $\iota \in \textsc{Inst}'$ and abstract states $\sigma_1, \sigma_2 \in \textsc{Map}$, if $\sigma_1 \sqsubseteq \sigma_2$ then $\textsc{flow}[\![\iota]\!](\sigma_1) \sqsubseteq \textsc{flow}[\![\iota]\!](\sigma_2)$. It can also be shown that the flow function is locally sound, *i.e.* the flow function models the concrete semantics at each

$$
\begin{aligned}
\textsc{flow}(x \texttt{ := } y, \sigma) &= \sigma[x \mapsto \sigma(y)] \\
\textsc{flow}(\texttt{branch } x, \sigma) &= \sigma \\
\textsc{flow}(\texttt{if } x, \sigma) &= \sigma[x \mapsto \texttt{NonNull}] \\
\textsc{flow}(\texttt{else } x, \sigma) &= \sigma[x \mapsto \texttt{Null}] \\
\textsc{flow}(x \texttt{ := } m\texttt{@}a(y\texttt{@}b), \sigma) &= \sigma[x \mapsto a] \\
\textsc{flow}(\texttt{proc } m\texttt{@}a(y\texttt{@}b), \sigma) &= \sigma_0[y \mapsto b] \\
\textsc{flow}(x \texttt{ := null}, \sigma) &= \sigma[x \mapsto \texttt{Null}] \\
\textsc{flow}(x \texttt{ := new}(\overline{f}), \sigma) &= \sigma[x \mapsto \texttt{NonNull}] \\
\textsc{flow}(x \texttt{ := } y \wedge z, \sigma) &= \begin{cases} \sigma[x \mapsto \texttt{Null}] & \text{if } \texttt{Null} \in \{\sigma(y), \sigma(z)\} \\ \sigma[x \mapsto \texttt{Nullable}] & \text{if } \texttt{Nullable} \in \{\sigma(y), \sigma(z)\} \\ \sigma[x \mapsto \texttt{NonNull}] & \text{otherwise} \end{cases} \\
\textsc{flow}(x \texttt{ := } y \vee z, \sigma) &= \begin{cases} \sigma[x \mapsto \texttt{NonNull}] & \text{if } \texttt{NonNull} \in \{\sigma(y), \sigma(z)\} \\ \sigma[x \mapsto \texttt{Nullable}] & \text{if } \texttt{Nullable} \in \{\sigma(y), \sigma(z)\} \\ \sigma[x \mapsto \texttt{Null}] & \text{otherwise} \end{cases} \\
\textsc{flow}(x \texttt{ := } y.f, \sigma) &= \sigma[x \mapsto \texttt{Nullable}][y \mapsto \texttt{NonNull}] \\
\textsc{flow}(x.f \texttt{ := } y, \sigma) &= \sigma[x \mapsto \texttt{NonNull}] \\
\textsc{flow}(\texttt{main}, \sigma) &= \sigma_0
\end{aligned}
$$

Figure 7 All consequential cases of the flow function used by NPA.

step. To express this property formally, we define the predicate $\textsc{desc}(\rho, \sigma)$ on $\textsc{Env} \times \textsc{Map}$, which says that the abstract state σ "describes" the concrete environment ρ:

$$\textsc{desc}(\rho, \sigma) \iff \text{for all } x \in \textsc{Var} \,.\, \rho(x) \in \textsc{conc}(\sigma(x)).$$

Then, if $\langle S' \cdot \langle \rho, [\iota]_v \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \rho', v' \rangle \cdot S \parallel \mu' \rangle$, it must be the case that

$$\textsc{desc}(\rho, \sigma) \implies \textsc{desc}(\rho', \textsc{flow}[\![\iota]\!](\sigma)) \quad \text{for all} \quad \sigma \in \textsc{Map}.$$

4.3 Fixpoint Algorithm

This brings us to Algorithm 1 [16], which is used to analyze a program and compute whether each program variable is `Nullable`, `NonNull`, or `Null` at each program point (the program results π). More specifically, the algorithm applies the flow function to each program instruction recording or updating the results until a fixpoint is reached – *i.e.* until the results stop changing (becoming more approximate). The algorithm will always reach a fixpoint (terminate), because FLOW is monotone and the height of the semilattice (Sec. 4.1) is finite. Note, the algorithm does not specify the order in which instructions are analyzed, because the order does not affect the results when FLOW is monotonic. An implementation may choose to analyze instructions in CFG order – following the directed edges of the CFG.

4.4 Safety Function & Static Warnings

Next, we present a way to use analysis results π produced by the fixpoint algorithm to determine whether to accept or reject a given program. Our goal is to ensure that when

Algorithm 1 Kildall's worklist algorithm.

```
1:  function KILDALL(FLOW, ⊔, p)
2:      π ← {v ↦ ∅ : v ∈ VERT_p}
3:      V ← VERT_p                                   ▷ V ⊆ VERT_p
4:      while V ≠ ∅ do
5:          [ι]_v ← an element of V                  ▷ v ∈ V and ι = INST_p(v)
6:          V ← V \ {v}                              ▷ v ∉ V
7:          σ ← π(v)
8:          σ' ← FLOW⟦ι⟧(σ)
9:          for v →p u do                            ▷ u ∈ VERT_p
10:             if σ' ⊔ π(u) ≠ π(u) then             ▷ think of as σ' ⋢ π(u)
11:                 π(u) ← π(u) ⊔ σ'
12:                 V ← V ∪ {u}
13:             end if
14:         end for
15:     end while
16:     return π
17: end function
```

$$\begin{aligned}
\textsc{safe}(x \texttt{ := } m\texttt{@}a(y\texttt{@}b), y) &= b \\
\textsc{safe}(\texttt{return } y\texttt{@}a, y) &= a \\
\textsc{safe}(x \texttt{ := } y.f, y) &= \texttt{NonNull} \\
\textsc{safe}(x.f \texttt{ := } y, x) &= \texttt{NonNull}
\end{aligned}$$

Figure 8 All nontrivial cases of the safety function.

we run the program, it will not get stuck; that is, for any state ξ that the program reaches, we want to ensure that either ξ is a final state $\langle E \cdot \mathsf{nil} \parallel \mu \rangle$ or there is another state ξ' such that $\xi \longrightarrow_p \xi'$. To do this, we define the safety function $\textsc{safe}[\![\iota]\!](x) : \textsc{Inst}' \times \textsc{Var} \to \textsc{Abst}$, which returns the abstract value representing the set of "safe" values x can take on before ι is executed. Figure 8 gives a few representative cases for SAFE, and in all the cases not shown SAFE returns `Nullable`. In particular, a procedure call's argument must adhere to the procedure's parameter annotation, a return value must adhere to its corresponding return annotation, and all field accesses must have non-null receivers. Therefore, the safety function guards against all undefined behavior.

4.4.1 Static Warnings

Now, we can state the meaning of a valid program $p \in \textsc{Prog}'$:

$$\text{for all} \quad [\iota]_v \in \textsc{Vert}_p \quad \text{and} \quad x \in \textsc{Var}\,. \quad \pi(v) = \sigma \quad \Longrightarrow \quad \sigma(x) \sqsubseteq \textsc{safe}[\![\iota]\!](x)$$

$$\text{where} \quad \pi = \textsc{Kildall}(\textsc{flow}, \sqcup, p).$$

That is, NPA emits static warnings when the fixpoint results disagree, according to the partial order $\sqsubseteq$, with the safety function. Also, we prove in Section 4.5 that a valid program does not get stuck.

4.5 Soundness of NPA

As discussed above, PICL's semantics are designed to get stuck when procedure annotations are violated or when null objects are dereferenced. Therefore, informally *soundness* says that a valid program does not get stuck during execution. Formally, soundness is defined with progress and preservation statements. Before their statement we must first define the notion of valid states to complement our definition of valid programs:

Let $p \in \textsc{Prog}'$. A state $\xi = \langle\langle \rho_1, v_1\rangle \cdot \langle \rho_2, v_2\rangle \cdots \langle \rho_n, v_n\rangle \cdot \mathsf{nil} \parallel \mu\rangle \in \textsc{State}_p$ is *valid* if

for all $1 \le i \le n$. $\textsc{desc}(\rho_i, \pi(v_i))$ where $\pi = \textsc{Kildall}(\textsc{flow}, \sqcup, p)$.

A state is *valid* if it is described by the static analysis results π.

▶ **Proposition 1** (static progress). *Let $p \in \textsc{Prog}'$ be valid. If $\xi = \langle E_1 \cdot E_2 \cdot S \parallel \mu\rangle \in \textsc{State}_p$ is valid then $\xi \longrightarrow_p \xi'$ for some $\xi' \in \textsc{State}_p$.*

▶ **Proposition 2** (static preservation). *Let $p \in \textsc{Prog}'$ be valid. If $\xi \in \textsc{State}_p$ is valid and $\xi \longrightarrow_p \xi'$ then ξ' is valid.*

5 Gradual Null-Pointer Analysis

In this section, we derive GNPA from NPA, presented previously (Sec. 4). We proceed following the Abstracting Gradual Typing methodology introduced by Garcia *et al.* [14] in the context of gradual type systems, adapting it to fit the concepts of static analysis.

We present the GNPA's lifted semilattice (Sec. 5.1), flow and safety functions (Sec. 5.2), and fixpoint algorithm (Sec. 5.3). We also discuss how static (Sec. 5.4) and run-time warnings (Sec. 5.5) are generated by the analysis. Finally, Section 5.6 establishes the main properties of GNPA.

Note, here, annotations may be missing, so we extend our set of annotations with `?`: $\textsc{Ann} = \{\texttt{NonNull}, \texttt{Nullable}\} \cup \{\texttt{?}\}$.

5.1 Lifting the Semilattice

In this section, we lift the semilattice $(\textsc{Abst}, \sqsubseteq, \sqcup)$ (Sec. 4.1) by following the Abstracting Gradual Typing (AGT) framework [14]. First, we extend the set of semilattice elements $\textsc{Abst}$ to the new set $\widetilde{\textsc{Abst}} \supseteq \textsc{Abst}$:

$$\widetilde{\textsc{Abst}} = \textsc{Abst} \cup \{\texttt{?}\} \cup \{a\texttt{?} : a \in \textsc{Abst}\} =$$

$$\{\texttt{Nullable}, \texttt{NonNull}, \texttt{Null}, \texttt{?}, \texttt{NonNull?}, \texttt{Null?}\}.$$

Note that we equate the elements `Nullable?` and `Nullable` in $\widetilde{\textsc{Abst}}$. In Section 5.1.1, we give the semantics of the new lattice elements resulting in $\top = \texttt{Nullable?} = \texttt{Nullable}$. If $\textsc{Abst}$ had a bottom element $\bot$, then $\bot = \bot\texttt{?}$ similarly.

The join $\sqcup$ and partial order $\sqsubseteq$ are also lifted to their respective counterparts $\widetilde{\sqcup}$ (Sec. 5.1.2) and $\widetilde{\sqsubseteq}$ (Sec. 5.1.3). The resulting lifted semilattice $(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$ with lifted relation $\widetilde{\sqsubseteq}$ underpins the optimism in GNPA.

5.1.1 Giving Meaning to Missing Annotations

A straightforward way to handle `?` would be to make it the top element $? = \top$ or the bottom element $? = \bot$ of NPA's semilattice. However, neither choice is sufficient for our goal:

- If $? = \bot$, then $? \sqsubseteq a$ for all $a \in \textsc{Abst}$ and $\textsc{conc}(\bot) = \varnothing$. As a result, if the return annotation of a procedure was `?`, then we could use the return value in any context without the analysis giving a warning. But, anytime an initialized variable is checked against the `?` annotation, such as checking the non-null return value y against the `?` return annotation `NonNull` $\sqsubseteq$ `?`, the check will fail as $a \not\sqsubseteq ?$ for all $a \in \textsc{Abst} \,.\, a \neq \bot$.
- If we let $? = \top$ then we have $a \sqsubseteq ?$ for all $a \in \textsc{Abst}$. Therefore, we can pass any argument to a parameter annotated as `?` without the static part of GNPA giving a warning. But, if the return annotation of that procedure is `?`, then the analysis will produce false positives in caller contexts wherever the return value is dereferenced. In other words, our analysis would operate exactly as `PolyNull` for the example in Fig. 1, which is not ideal.

Our goal is to construct an analysis system that does not produce false positive static warnings when a developer omits an annotation. To achieve this, we draw on work in gradual typing [14]. We define the injective concretization function $\gamma : \widetilde{\textsc{Abst}} \to \mathcal{P}^+(\textsc{Abst})$ where $\widetilde{\textsc{Abst}} \supseteq \textsc{Abst}$ is the lifted semilattice element set (Sec. 5.1):

$$\gamma(a) = \{a\} \quad \text{for} \quad a \in \textsc{Abst}, \quad \gamma(?) = \textsc{Abst}, \quad \text{and} \quad \gamma(a?) = \{b \in \textsc{Abst} : a \sqsubseteq b\}.$$

An element in $\textsc{Abst}$ is mapped to itself as it can only represent itself. In contrast, `?` may represent any element in $\textsc{Abst}$ at all times to support optimism in all possible contexts. Further, $a?$ means "a or anything more general than it," in contrast to a gradual formula $\phi \wedge ?$ that means "ϕ or anything more specific than it" [2]. As a result, $a?$ does not play the intuitive role of "supplying missing information," as it would in gradual verification. Instead, $a?$ is simply an artifact of our construction, which is why the only element of $\textsc{Ann} \setminus \textsc{Abst}$ is `?`.

Then, if $\gamma(\widetilde{a}) \subseteq \gamma(\widetilde{b})$ for some $\widetilde{a}, \widetilde{b} \in \widetilde{\textsc{Abst}}$, we write $\widetilde{a} \lesssim \widetilde{b}$ and say that $\widetilde{a}$ *is more precise than* $\widetilde{b}$. Further, $\iota_1 \lesssim \iota_2$ means that 1) the two instructions are equal except for their annotations, and 2) the annotations in ι_1 are more precise than the corresponding annotations in ι_2.

5.1.2 Lifted Join $\widetilde{\sqcup}$

We begin by introducing a semilattice definition [9], which states that a semilattice is an algebraic structure $(S, \sqcup)$ where for all $x, y, z \in S$ the following hold:

- $x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$ (associativity)
- $x \sqcup y = y \sqcup x$ (commutativity)
- $x \sqcup x = x$ (idempotency)

Then, we write $x \sqsubseteq y$ when $x \sqcup y = y$ and it can be shown this $\sqsubseteq$ is a partial order. Recall that NPA uses $\sqcup$ in Algorithm 1 to compute a fixpoint that describes the behavior of a program p. The fixpoint can only be reached when $\sqcup$ is idempotent. Similarly, $\sqcup$ must be commutative and associative so that program instructions can be analyzed in any order. Thus, our extended join operation $\widetilde{\sqcup} : \widetilde{\textsc{Abst}} \times \widetilde{\textsc{Abst}} \to \widetilde{\textsc{Abst}}$ must be associative, commutative, and idempotent making $(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$ a join-semilattice.

To define such a function we turn to insights from gradual typing [14]. We define an abstraction function $\alpha : \mathcal{P}^+(\textsc{Abst}) \to \widetilde{\textsc{Abst}}$, which forms a Galois connection with γ:

$$\alpha(\widehat{a}) = \gamma^{-1}\left(\bigcap_{\substack{\widetilde{b} \in \widetilde{\textsc{Abst}} \\ \gamma(\widetilde{b}) \supseteq \widehat{a}}} \gamma(\widetilde{b})\right)$$

where, for $a \in \textsc{Abst}$, γ^{-1} is:

$$\gamma^{-1}(\{a\}) = a \qquad \gamma^{-1}(\textsc{Abst}) = \texttt{?} \qquad \gamma^{-1}(\{b \in \textsc{Abst} : a \sqsubseteq b\}) = a\texttt{?}.$$

Then we define the join of $\widetilde{a}, \widetilde{b} \in \widetilde{\textsc{Abst}}$ as follows:

$$\widetilde{a} \mathbin{\widetilde{\sqcup}} \widetilde{b} = \alpha(\{a \sqcup b : \ a \in \gamma(\widetilde{a}) \text{ and } b \in \gamma(\widetilde{b})\})$$

For example,

$$\begin{aligned}
\texttt{NonNull} \mathbin{\widetilde{\sqcup}} \texttt{?} &= \alpha(\{a \sqcup b : a \in \{\texttt{NonNull}\} \text{ and } b \in \textsc{Abst}\}) && (1)\\
&= \alpha(\{\texttt{NonNull}, \ \texttt{Nullable}\}) && (2)\\
&= \gamma^{-1}\left(\gamma(\texttt{NonNull?}) \cap \gamma(\texttt{?})\right) && (3)\\
&= \gamma^{-1}\left(\{\texttt{NonNull}, \ \texttt{Nullable}\} \cap \textsc{Abst}\right) && (4)\\
&= \gamma^{-1}\left(\{\texttt{NonNull}, \ \texttt{Nullable}\}\right) && (5)\\
&= \texttt{NonNull?} && (6)
\end{aligned}$$

That is, the join of all the Abst elements represented by `NonNull` and `?` results in the set {`NonNull`, `Nullable`} (1, 2). Applying α to this set is equivalent to applying γ^{-1} to $\gamma(\texttt{NonNull?}) \cap \gamma(\texttt{?})$ (3); because, the only $\widetilde{\textsc{Abst}}$ elements that represent both `NonNull` and `Nullable` are `NonNull?` and `?`. The intersection of $\gamma(\texttt{NonNull?})$ and $\gamma(\texttt{?})$ is {`NonNull`, `Nullable`} (4, 5), so we are really applying γ^{-1} to {`NonNull`, `Nullable`} (5). Therefore, $\texttt{NonNull} \mathbin{\widetilde{\sqcup}} \texttt{?} = \texttt{NonNull?}$ (6). Notice, the intersection of the representative sets $\gamma(\texttt{NonNull?})$ and $\gamma(\texttt{?})$ of $\{\texttt{NonNull}, \ \texttt{Nullable}\} = \widehat{a}$ is used to find the most precise element in $\widetilde{\textsc{Abst}}$ that can represent $\widehat{a}$.

Now we return to the properties of $\widetilde{\sqcup}$. Since $\sqcup$ is commutative, we have that $\widetilde{\sqcup}$ is commutative. Idempotency is also not too onerous: it is equivalent to the condition that every element of $\widetilde{\textsc{Abst}}$ represents a subsemilattice of Abst. That is, for every $\widetilde{a} \in \widetilde{\textsc{Abst}}$ and $a_1, a_2 \in \gamma(\widetilde{a})$, we must have $a_1 \sqcup a_2 \in \gamma(\widetilde{a})$. This is true by construction. Associativity is tricky and motivates our complex definition of $\widetilde{\textsc{Abst}}$. Ideally, $\widetilde{\textsc{Abst}}$ would be defined simply as $\textsc{Abst} \cup \{\texttt{?}\}$, however in this case $\widetilde{\sqcup}$ is not associative:

$$\begin{aligned}
\texttt{Null} \mathbin{\widetilde{\sqcup}} (\texttt{NonNull} \mathbin{\widetilde{\sqcup}} \texttt{?}) &= \texttt{Null} \mathbin{\widetilde{\sqcup}} \texttt{?}\\
&= \texttt{?}\\
&\neq \texttt{Nullable}\\
&= \texttt{Nullable} \mathbin{\widetilde{\sqcup}} \texttt{?}\\
&= (\texttt{Null} \mathbin{\widetilde{\sqcup}} \texttt{NonNull}) \mathbin{\widetilde{\sqcup}} \texttt{?}.
\end{aligned}$$

Fortunately, our definition of $\widetilde{\textsc{Abst}}$ which also includes the intermediate optimistic elements `NonNull?` and `Null?` results in an associative $\widetilde{\sqcup}$ function and a finite-height semilattice $(\widetilde{\textsc{Abst}}, \ \widetilde{\sqcup})$. Figure 9 shows the semilattice structure induced by $\widetilde{\sqcup}$.

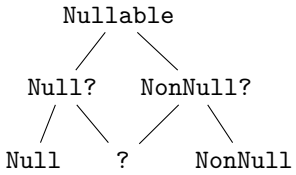


Figure 9 The semilattice structure induced by the lifted join $\widetilde{\sqcup}$. Specifically, this is the Hasse diagram of the partial order $\{(\widetilde{a}, \widetilde{b}) : \widetilde{a} \,\widetilde{\sqcup}\, \widetilde{b} = \widetilde{b}\}$.

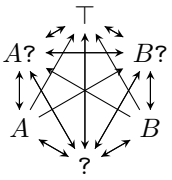


Figure 10 The lifted partial order, where each directed edge $\widetilde{a} \to \widetilde{b}$ means $\widetilde{a} \,\widetilde{\sqsubseteq}\, \widetilde{b}$. (Self-loops are omitted). Here, `Nullable` is abbreviated $\top$, and `Null` and `NonNull` are abbreviated A and B respectively.

5.1.3 Lifted Order $\widetilde{\sqsubseteq}$

Now it is fairly straightforward to construct $\widetilde{\sqsubseteq}$. Recall, NPA emits static warnings when the fixpoint results disagree with the safety function, according to the partial order $\sqsubseteq$. The fixpoint results and the safety function now return elements in $\widetilde{\textsc{Abst}}$, so we lift $\sqsubseteq$ to $\widetilde{\sqsubseteq} \subseteq \widetilde{\textsc{Abst}} \times \widetilde{\textsc{Abst}}$ using the concretization function γ:

$$\widetilde{a} \,\widetilde{\sqsubseteq}\, \widetilde{b} \quad \iff \quad \exists\,. \quad a \in \gamma(\widetilde{a}) \quad \text{and} \quad b \in \gamma(\widetilde{b}) \quad \text{such that} \quad a \sqsubseteq b \quad \text{for} \quad \widetilde{a}, \widetilde{b} \in \widetilde{\textsc{Abst}}.$$

Figure 10 gives the lifted order relation $\widetilde{\sqsubseteq}$ in graphical form.

The $\widetilde{\sqsubseteq}$ predicate is a maximally permissive version of the $\sqsubseteq$ predicate for `NonNull?`, `Null?`, and `?`. For example, `?` $\widetilde{\sqsubseteq}$ `NonNull` since γ(`?`) = {`NonNull`, `Null`, `Nullable`}, γ(`NonNull`) = {`NonNull`}, and `NonNull` $\sqsubseteq$ `NonNull`. By similar reasoning, `NonNull` $\widetilde{\sqsubseteq}$ `?`. In fact, `?` $\widetilde{\sqsubseteq}$ a $\widetilde{\sqsubseteq}$ `?`, `NonNull?` $\widetilde{\sqsubseteq}$ a $\widetilde{\sqsubseteq}$ `NonNull?`, and `Null?` $\widetilde{\sqsubseteq}$ a $\widetilde{\sqsubseteq}$ `Null?` for $a \in \textsc{Abst}$. So, clearly $\widetilde{\sqsubseteq}$ is not a partial order. The $\widetilde{\sqsubseteq}$ predicate must be maximally permissive to support the optimism used in the `safeReverse` example from Figure 1 (Sec. 2.2): calls to `safeReverse` with null and non-null arguments are valid and dereferences of its return values are also valid. However, $\widetilde{\sqsubseteq}$ is the same as $\sqsubseteq$ when both of its arguments come from $\textsc{Abst}$, *e.g.* `NonNull` $\widetilde{\sqsubseteq}$ `Nullable` and `Nullable` $\widetilde{\not\sqsubseteq}$ `NonNull`. This allows our gradual analysis to apply NPA where annotations are complete enough to support it.

5.1.4 Properties

We previously mentioned some of the properties which $(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$ satisfy. Here, we formally state them, and their proofs can be found in the appendix of the full version of this paper [10].

▶ **Proposition 3.** *$(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$ is a semilattice; in other words, $\widetilde{\sqcup}$ is associative, idempotent, and commutative.*

▶ **Proposition 4.** *If the height of $(\textsc{Abst}, \sqcup)$ is $n > 0$, then the height of $(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$ is $n + 1$ (in particular, $(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$ has finite height).*

5.2 Lifting the Flow & Safety Functions

Now both instructions and abstract states ($\widetilde{\sigma} \in \widetilde{\textsc{Map}} = \textsc{Var} \rightharpoonup \widetilde{\textsc{Abst}}$) may contain optimistic abstract values. Therefore, similar to lifting the join $\widetilde{\sqcup}$, we follow the AGT *consistent function lifting* approach [14] when defining GNPA's flow function $\widetilde{\textsc{flow}} : \textsc{Inst} \times \widetilde{\textsc{Map}} \to \widetilde{\textsc{Map}}$ for this new domain.

Specifically, for $\iota \in \textsc{Inst}$ and $\widetilde{\sigma} = \{x \mapsto \widetilde{a}_x : x \in \textsc{Var}\} \in \widetilde{\textsc{Map}}$, we define

$$\widetilde{\textsc{flow}}[\![z \texttt{ := } m\texttt{@}a(y\texttt{@}b)]\!](\widetilde{\sigma}) = \{x \mapsto \alpha(\{(\textsc{flow}[\![z \texttt{ := } m\texttt{@}a'(y\texttt{@}b')]\!](\sigma'))(x) \\ : a' \in \gamma(a) \wedge b' \in \gamma(b) \wedge \sigma' \in \Sigma\}) : x \in \textsc{Var}\}$$
$$\widetilde{\textsc{flow}}[\![\texttt{proc } m\texttt{@}a(y\texttt{@}b)]\!](\widetilde{\sigma}) = \{x \mapsto \alpha(\{(\textsc{flow}[\![\texttt{proc } m\texttt{@}a'(y\texttt{@}b')]\!](\sigma'))(x) \\ : a' \in \gamma(a) \wedge b' \in \gamma(b) \wedge \sigma' \in \Sigma\}) : x \in \textsc{Var}\}$$
$$\widetilde{\textsc{flow}}[\![\iota]\!](\widetilde{\sigma}) = \{x \mapsto \alpha(\{(\textsc{flow}[\![\iota]\!](\sigma'))(x) : \sigma' \in \Sigma\}) : x \in \textsc{Var}\} \quad \text{otherwise}$$

where $\Sigma = \{\{x \mapsto a_x : x \in \textsc{Var}\} : a_x \in \gamma(\widetilde{a}_x) \text{ for all } x \in \textsc{Var}\}$.

Note that the procedure call and procedure entry instructions are the only instructions in FLOW's domain that may contain `?` annotations, so the corresponding FLOW rules are lifted with respect to those annotations. Similarly, all rules are lifted with respect to their abstract states.

Recall that we defined the predicate DESC on $\textsc{Env} \times \textsc{Map}$ to express the local soundness of FLOW. For $\widetilde{\textsc{flow}}$, we lift DESC to $\widetilde{\textsc{desc}}$ on $\textsc{Env} \times \widetilde{\textsc{Map}}$ such that it is maximally permissive like the $\widetilde{\sqsubseteq}$ predicate:

$$\widetilde{\textsc{desc}}(\rho, \widetilde{\sigma}) \iff \textsc{desc}(\rho, \sigma) \text{ for some } \sigma \in \Sigma$$

where Σ is constructed in the same way as for $\widetilde{\textsc{flow}}$.

Finally, we again follow the consistent function lifting methodology to construct $\widetilde{\textsc{safe}} : \textsc{Inst} \times \textsc{Var} \to \widetilde{\textsc{Abst}}$ from $\textsc{safe} : \textsc{Inst}' \times \textsc{Var} \to \textsc{Abst}$:

$$\widetilde{\textsc{safe}}[\![z \texttt{ := } m\texttt{@}a(y\texttt{@}b)]\!](x) = \alpha(\{\textsc{safe}[\![z \texttt{ := } m\texttt{@}a'(y\texttt{@}b')]\!](x) : a' \in \gamma(a) \wedge b' \in \gamma(b)\})$$
$$\widetilde{\textsc{safe}}[\![\texttt{proc } m\texttt{@}a(y\texttt{@}b)]\!](x) = \alpha(\{\textsc{safe}[\![\texttt{proc } m\texttt{@}a'(y\texttt{@}b')]\!](x) : a' \in \gamma(a) \wedge b' \in \gamma(b)\})$$
$$\widetilde{\textsc{safe}}[\![\texttt{return } y\texttt{@}a]\!](x) = \alpha(\{\textsc{safe}[\![\texttt{return } y\texttt{@}a']\!](x) : a' \in \gamma(a)\})$$
$$\widetilde{\textsc{safe}}[\![\iota]\!](x) = \alpha(\textsc{safe}[\![\iota]\!](x)) \quad \text{otherwise}$$

Other than the casewise-defined FLOW rules for $\wedge$ and $\vee$, the lifted $\widetilde{\textsc{flow}}$ and $\widetilde{\textsc{safe}}$ functions simplify down to the same computation rules as FLOW and SAFE as shown in Figure 7 and Figure 8 respectively, replacing FLOW with $\widetilde{\textsc{flow}}$ and SAFE with $\widetilde{\textsc{safe}}$.

5.3 Lifting the Fixpoint Algorithm

To lift the fixpoint algorithm, we simply plug $\widetilde{\textsc{flow}}$ and $\widetilde{\sqcup}$ into Algorithm 1 to compute $\widetilde{\pi} = \textsc{Kildall}(\widetilde{\textsc{flow}}, \widetilde{\sqcup}, p) : \textsc{Vert}_p \to \widetilde{\textsc{Map}}$ for any $p \in \textsc{Prog}$.

5.4 Static Warnings

Using the lifted safety function, we say that a partially-annotated program $p \in \textsc{Prog}$ is *statically valid* if

$$\text{for all } [\iota]_v \in \textsc{Vert}_p \text{ and } x \in \textsc{Var}, \quad \widetilde{\pi}(v) = \widetilde{\sigma} \implies \widetilde{\sigma}(x) \mathrel{\widetilde{\sqsubseteq}} \widetilde{\textsc{safe}}[\![\iota]\!](x)$$

where $\widetilde{\pi} = \textsc{Kildall}(\widetilde{\textsc{flow}}, \widetilde{\sqcup}, p)$.

Each piece of GNPA's static system ($(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$, $\widetilde{\sqsubseteq}$, $\widetilde{\textsc{flow}}$, $\widetilde{\textsc{safe}}$, and the fixpoint algorithm) is designed to be maximally optimistic for missing annotations. Therefore, the resulting system will not produce false positive warnings due to missing annotations. The system is also designed to apply NPA where annotations are available to support it, so it will still warn about violations of procedure annotations or null object dereferences where possible. See Section 2.2 for more information.

5.5 Dynamic Checking

GNPA's static system reduces false positive warnings at the cost of soundness. For example, as in Section 2.3, the analysis may assume a variable with a `?` annotation is non-null to satisfy an object dereference when the variable is actually null. In order to avoid false negatives and ensure that our gradual analysis is sound, we modify the semantics of PICL to insert run-time checks where the analysis may be unsound. That is, if p is *statically valid* and there are program points $[\iota]_v$ such that

$$a \not\sqsubseteq \bigsqcup \gamma(\widetilde{\textsc{safe}}[\![\iota]\!](x)) \quad \text{for some} \quad x \in \textsc{Var} \quad \text{and} \quad a \in \gamma((\widetilde{\pi}(v))(x)),$$

then a run-time check must be inserted at those points to ensure the value of x is in $\textsc{conc}(\bigsqcup \gamma(\widetilde{\textsc{safe}}[\![\iota]\!](x)))$.

More precisely, we define a dedicated error state `error` and expand the set of run-time states to be $\widetilde{\textsc{State}}_p = \textsc{State}_p \cup \{\texttt{error}\}$. Then we define a restricted semantics $\widetilde{\longrightarrow}_p$ on $\widetilde{\textsc{State}}_p \times \widetilde{\textsc{State}}_p$ as follows. Let $\xi \in \textsc{State}_p$. If

$$\xi = \langle\langle \rho, [\iota]\rangle \cdot S \parallel \mu\rangle \quad \text{and} \quad \neg\widetilde{\textsc{desc}}(\rho, \{x \mapsto \widetilde{\textsc{safe}}[\![\iota]\!](x) : x \in \textsc{Var}\})$$

then $\xi \mathrel{\widetilde{\longrightarrow}_p} \texttt{error}$. If there is some $\xi' \in \textsc{State}_p$ such that $\xi \longrightarrow_p \xi'$, then $\xi \mathrel{\widetilde{\longrightarrow}_p} \xi'$. Otherwise, there is no $\widetilde{\xi}' \in \widetilde{\textsc{State}}_p$ such that $\xi \mathrel{\widetilde{\longrightarrow}_p} \xi'$.

5.6 Gradual Properties

GNPA is sound, *conservative extension* of NPA – the static system is applied in full to programs with complete annotations, and adheres to the gradual guarantees inspired by Siek *et al.* [23]. The gradual guarantees ensure losing precision is harmless, *i.e.* increasing the number of missing annotations in a program does not break its validity or reducibility.

To formally present each property, we first extend the notion of a valid state. Let $p \in \textsc{Prog}$. A state $\xi = \langle\langle \rho_1, v_1\rangle \cdot \langle \rho_2, v_2\rangle \cdots \langle \rho_n, v_n\rangle \cdot \mathsf{nil} \parallel \mu\rangle \in \textsc{State}_p$ is valid if

$$\text{for all} \quad 1 \le i \le n, \quad \widetilde{\textsc{desc}}(\rho_i, \widetilde{\pi}(v_i)) \quad \text{where} \quad \widetilde{\pi} = \textsc{Kildall}(\widetilde{\textsc{flow}}, \widetilde{\sqcup}, p).$$

Then, for fully-annotated programs, GNPA and the modified semantics are conservative extensions of NPA and PICL's semantics, respectively.

▶ **Proposition 5** (conservative static extension)**.**
If $p \in \textsc{Prog}'$ *then* $\textsc{Kildall}(\textsc{flow}, \sqcup, p) = \textsc{Kildall}(\widetilde{\textsc{flow}}, \widetilde{\sqcup}, p)$.

▶ **Proposition 6** (conservative dynamic extension)**.** *Let* $p \in \textsc{Prog}'$ *be valid, and let* $\xi_1, \xi_2 \in \textsc{State}_p$. *If* ξ_1 *is valid then* $\xi_1 \longrightarrow_p \xi_2$ *if and only if* $\xi_1 \mathrel{\widetilde{\longrightarrow}_p} \xi_2$.

GNPA is sound, *i.e.* valid programs will not get stuck during execution. However, programs may step to a dedicated `error` state when run-time checks fail. Soundness is stated with a progress and preservation argument.

▶ **Proposition 7** (gradual progress). *Let $p \in \textsc{Prog}$ be valid. If $\xi = \langle E_1 \cdot E_2 \cdot S \parallel \mu \rangle \in \textsc{State}_p$ is valid then $\xi \mathrel{\widetilde{\longrightarrow}_p} \widetilde{\xi'}$ for some $\widetilde{\xi'} \in \widetilde{\textsc{State}_p}$.*

▶ **Proposition 8** (gradual preservation). *Let $p \in \textsc{Prog}$ be valid. If $\xi \in \textsc{State}_p$ is valid and $\xi \mathrel{\widetilde{\longrightarrow}_p} \xi'$ for some $\xi' \in \textsc{State}_p$, then ξ' is valid.*

Finally, GNPA satisfies both the static and dynamic gradual guarantees. Both of the guarantees rely on a definition of *program precision*. Specifically, if programs p_1 and p_2 are identical except perhaps that some annotations in p_2 are `?` where they are not `?` in p_1, then we say that p_1 *is more precise than* p_2, and write $p_1 \lesssim p_2$.

Then, the *static gradual guarantee* states that increasing the number of missing annotations in a valid program does not introduce static warnings (*i.e.* break program validity).

▶ **Proposition 9** (static gradual guarantee). *Let $p_1, p_2 \in \textsc{Prog}$ such that $p_1 \lesssim p_2$. If p_1 is statically valid then p_2 is statically valid.*

The *dynamic gradual guarantee* ensures that increasing the number of missing annotations in a program does not change the observable behavior of the program (*i.e.* break program reducibility for valid programs).

▶ **Proposition 10** (dynamic gradual guarantee). *Let $p_1, p_2 \in \textsc{Prog}$ be statically valid, where $p_1 \lesssim p_2$. Let $\xi_1, \xi_2 \in \textsc{State}_{p_2}$. If $\xi_1 \mathrel{\widetilde{\longrightarrow}_{p_1}} \xi_2$ then $\xi_1 \mathrel{\widetilde{\longrightarrow}_{p_2}} \xi_2$.*

Note, the small-step semantics $\widetilde{\longrightarrow}$ are designed to make the proofs of the aforementioned properties easier at the cost of easily implementable run-time checks. Therefore, we give the following proposition that connects a more implementable design to $\widetilde{\longrightarrow}$. That is, we can use the contrapositive of this proposition to implement more optimal run-time checks. Specifically, the naïve implementation would check each variable at each program point to make sure it satisfies the safety function for the instruction about to be executed. But Proposition 1 tells us that we only need to check variables at runtime when our analysis results don't already guarantee (statically) that they will satisfy the safety function.

▶ **Proposition 11** (run-time checks). *Let $p \in \textsc{Prog}$ be valid according to $\widetilde{\pi} = \textsc{Kildall}(\widetilde{\textsc{flow}}, \widetilde{\sqcup}, p)$, and let $\xi = \langle \langle \rho, [\iota]_v \rangle \cdot S \parallel \mu \rangle \in \textsc{State}_p$ be valid. If $\xi \mathrel{\widetilde{\longrightarrow}_p}$ `error` then there is some $x \in \textsc{Var}$ and $a \in \gamma((\widetilde{\pi}(v))(x))$ such that $a \not\sqsubseteq \bigsqcup \gamma(\widetilde{\textsc{safe}}[\![\iota]\!](x))$.*

6 Preliminary Empirical Evaluation

In this section, we discuss the implementation of GNPA and two studies designed to evaluate its usefulness in practice. Preliminary evidence suggests that our analysis can be used at scale, produces fewer false positives than state-of-the-art tools, and eliminates on average more than half of the null-pointer checks Java automatically inserts at run time. These results illustrate an important practical difference between GNPA and other null-pointer analyses. While a sound static analysis can be used to prove the redundancy of run-time checks, and an unsound static analysis can be used to reduce the number of false positives, neither of those can do both at the same time. On the other hand, GNPA can both prove the redundancy of run-time checks and reduce reported false positives

6.1 Research Questions

We seek answers to the following questions:

1. Can a gradual null-pointer analysis be effectively implemented and used at scale?

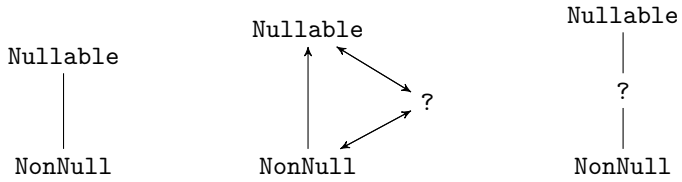


Figure 11 *Left:* The starting null-pointer semilattice for Graduator. *Middle:* The lifted partial ordering, where each directed edge $\widetilde{a} \to \widetilde{b}$ means $\widetilde{a} \mathrel{\widetilde{\sqsubseteq}} \widetilde{b}$. (Self-loops are omitted.) *Right:* The semilattice structure induced by the lifted join $\widetilde{\sqcup}$.

2. Does such a null-pointer analysis produce fewer false positives than industry-grade analyses?
3. Does the gradual null-pointer analysis perform significantly fewer null-pointer checks than the naïve approach of checking every dereference?

6.2 Prototype

Facebook Infer provides a framework to construct static analyses that use abstract interpretation. We built a prototype of GNPA, called *Graduator*, in this framework. Our prototype uses Infer's HIL intermediate language representation (IR). As a result, Graduator can be used to analyze code written in C, C++, Objective-C, and Java.

The preceding case study (Secs. 3–5) uses a base semilattice with three elements, `Null`, `NonNull`, and `Nullable`, in order to demonstrate that a semilattice lifting may contain additional intermediate optimistic elements, `Null?` and `NonNull?`. For simplicity, we implemented the semilattice from Figure 11, along with its lifted variant, order relation and join function, in our prototype. This semilattice is the same as the base one in the case study except it does not contain `Null`: the initial static semilattice has only `NonNull` and `Nullable`, and the gradual semilattice only adds one additional `?` element. There are a couple other differences between our formalism and our Graduator prototype, one of which is that Graduator allows field annotations while our formalism does not.

Infer does not support modifying Java source code, so Graduator simply reports the locations where it should insert run-time checks rather than inserting them directly. In fact, Graduator may output any of the following:

- `GRADUAL_STATIC` – a static warning.
- `GRADUAL_CHECK` – a location to check a possibly-null dereference.
- `GRADUAL_BOUNDARY` – another location to insert a check, such as passing an argument to a method, returning from a method, or assigning a value to a field.

Since Java checks for null-pointer dereferences automatically, soundness is preserved. A more complete implementation of GNPA would insert run-time checks as part of the build process. As a result, some bugs may be caught earlier when the gradual analysis inserts checks at method boundaries and field assignments.

By implementing Graduator with Infer's framework, Graduator is guaranteed to operate at scale. We also evaluate Graduator on a number of open source repositories as discussed in Sections 6.3 and 6.4. Thus, the answer to RQ1 is yes.

6.3 Static Warnings

To evaluate Graduator, we ran it on 15 of the 18 open-source Java repositories used to evaluate NullAway [3] (we excluded 3 of the 18 repositories because we were unable to

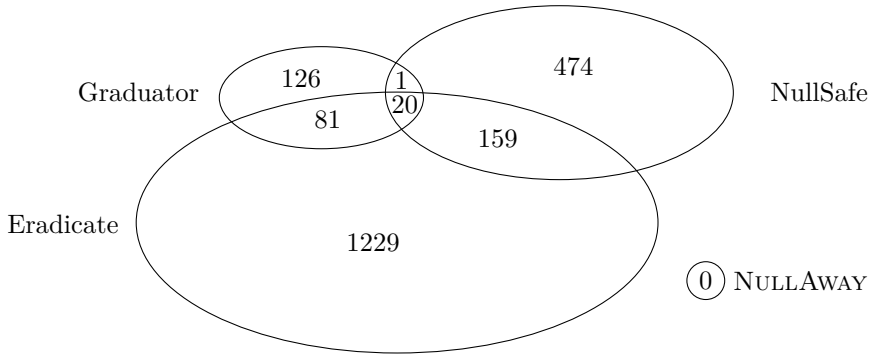


Figure 12 The total number of static warnings reported by the three Infer null checkers, for all 15 repositories.

successfully run Infer on them). We also ran NullAway, and Infer's existing null-pointer checkers Eradicate and NullSafe, on the repositories. Figure 12 shows the number of *static* warnings produced by each of these three checkers: 1489 for Eradicate, 654 for NullSafe, 228 for Graduator, and 0 for NullAway, for a total of 2371.

Based on the NullAway paper (in which Uber states that in practice they have found no instances of null-pointer dereferences caused by their tool's unsoundness), it seems reasonable to assume that these repositories do not have null-pointer bugs, since NullAway itself reports no static warnings for these repositories. After examining all 2371 warnings ourselves, we found that all but 57 (50 from Eradicate only, 2 from Graduator only, and 5 from Eradicate and Graduator but not NullSafe) were false positives due to systematic imprecision in the analysis tools. We were unable to determine whether the remaining 57 warnings represent actual bugs or not.

Under this assumption, Graduator reports significantly fewer false positives than Infer's existing null-pointer checkers (although in this respect, it is of course outperformed by NullAway) (RQ2). An interesting aspect of Figure 12 is how many warnings are produced by only one of the checkers: 1229 for Eradicate, 474 for NullSafe, and 126 for Graduator. Many of these warnings arose from generated and test case code.

6.3.1 Generated Code

Several of the 15 repositories generate code as part of their build process, and in some cases, the analysis tools gave warnings about the generated code. This accounts for

- 380 of the warnings given by NullSafe alone,
- 356 of the warnings given by Eradicate alone,
- 130 of the warnings given by both Eradicate and NullSafe but not Graduator, and
- 8 of the warnings given by Graduator alone.

Graduator reports significantly fewer static warnings for generated code, because such code is typically unannotated and Graduator is designed to be optimistic when annotations are missing.

6.3.2 Test Code

It is reasonable to assume that test code does not contain null dereference bugs, because if it did, then those bugs would show up when the tests are run. Static warnings about test code account for

Table 1 Percentage of null-dereference checks which Graduator found to be redundant.

repository	dereference sites	eliminated checks	percent eliminated
keyvaluestore	419	156	37%
uLeak	620	241	39%
butterknife	2773	1129	41%
jib	5896	2499	42%
skaffold-tools-for-java	366	185	51%
picasso	2719	1458	54%
meal-planner	858	475	55%
caffeine	9455	5701	60%
AutoDispose	3218	1993	62%
ColdSnap	6360	4325	68%
ReactiveNetwork	2097	1626	78%
okbuck	19089	15130	79%
FloatingActionButtonSpeedDial	3049	2581	85%
QRContact	1272	1171	92%
OANDAFX	2216	2056	93%
overall	60407	40726	67%

- 384 of the warnings given by Eradicate alone, and
- 73 of the warnings given by both Eradicate and Graduator, but not NullSafe.

That is, Graduator reports fewer warnings for test code than Eradicate, but more than NullSafe. The NullSafe checker does not appear to treat test code specially, so it is unclear why NullSafe is performing better than Graduator for such code.

6.3.3 Remaining False Positives

The reader may wonder why Graduator reports any false positives on this codebase, since it intuitively seems that the static portion of a gradual analysis ought to be optimistic. Examining the warnings given by Graduator, we see that none of the warnings are due to treating missing annotations pessimistically; instead, they are due to places where the analysis has whatever annotations it needs, but the analysis is imprecise in other respects. For example, one common source of false positives is when a field is checked for null, then is read again. Our original static analysis is limited in that it does not treat fields flow-sensitively, causing false positives that are independent of the choice to be gradual or not with respect to annotations.

NullAway avoids giving false positives on this same codebase, due to a combination of some unsound assumptions and a more precise analysis approach. While our approach for deriving gradual program analyses focuses on retaining soundness through a combination of static and dynamic checks, incorporating more precise analysis techniques (e.g. a flow-sensitive treatment of fields, perhaps in combination with a gradual alias analysis) could eliminate more of these false positives. In the meantime, our comparison to Eradicate and NullSafe is appropriate as these are the static analysis tools taking the most similar approach.

6.4 Run-time Checks

For the same set of 15 repositories analyzed by NullAway, we performed another experiment using our prototype. We configured Graduator to ignore *all* annotations, so in effect, every field, argument, and return value was annotated as `?`. For each repository, we counted all the locations where Graduator gave a `GRADUAL_STATIC`, `GRADUAL_CHECK`, or `GRADUAL_BOUNDARY`

warning, and compared that number to the total number of pointer dereferences in the code. By ignoring annotations, we ensured that each of these warnings appeared on dereferences, rather than allowing early checks at, e.g., method boundaries. We also ran analogous experiments with annotations enabled, but the number of run-time check warnings found were very similar to the numbers found with annotations disabled.

Table 1 shows what percentage of these dereference sites received no static warnings or run-time checks. Recall that Java automatically checks all dereferences to ensure that they are not null. Because GNPA is sound, this figure shows the percentage of null checks that are provably redundant, and could be safely removed by an ahead-of-time compiler.

Since we were able to eliminate an average of 67% of the null checks which Java automatically inserts, this experiment suggests the answer to RQ3 is yes. Note that these numbers only discuss the number of dereferences that appear in the code, and do not take into account which of these dereferences are executed more or less frequently at run-time.

This also illustrates an important practical difference between GNPA and other null-pointer analyses. While a sound static analysis can be used to prove the redundancy of run-time checks, and an unsound static analysis can be used to reduce the number of false positives, neither of those can do both at the same time. On the other hand, a gradual analysis can both prove the redundancy of run-time checks and reduce reported false positives.

7 Related Work

As discussed previously, our work builds on prior research in gradual typing: the criteria for gradual type systems [23] and the Abstracting Gradual Typing methodology, which develops a gradual type system from a purely static one [14]. In contrast to prior work in gradual typing, we address the challenges of tracking transitive dataflow relationships, rather than the local checks of typical type systems. In doing so, we gradualize, for the first time, the abstract interpretation of a program [8], and the canonical dataflow analysis fix-point algorithm [16].

The most closely related work in program analysis consists of *hybrid analyses*, which combine static and dynamic analysis techniques to counteract the weaknesses inherent to each approach. For example, Choi *et al.* [7] used a static analysis to substantially lower the run-time overhead of a dynamic data race analysis. Prior work on hybrid program analyses combines static and dynamic techniques in ad-hoc ways. Instead, we propose a principled methodology for deriving a hybrid (gradual) analysis from a static one, and show that the resulting analysis adheres to desirable properties such as soundness and the gradual guarantee.

There is a large body of literature on static program analysis, including multiple specialized conferences. Our work opens the door to gradual versions of them. Previously, we discussed existing null-pointer analysis tools [11], [3] and frameworks [21], and how GNPA is an improvement over them. Notably, our prototype is implemented in Infer's framework [11].

The Granullar type system [5] and the Blame for Null calculus[19] are gradual type systems for nullness, and thus solve a related problem to GNPA. The main difference in our work is that we use dataflow analysis instead of typing. This results in a significantly different user experience, as a full static specification within a gradual type system typically requires many more types to be specified (e.g. on all local variables) compared to a dataflow analysis, where for example we do not require (or even allow) nullity annotations on local variables. Basing our work on dataflow analysis also has a major impact on the technical development, requiring the novel lattice-based gradualization framework described in this

paper rather than the well-known type-based gradualization approaches used in Granullar and Blame for Null. Blame for Null also investigates the notion of blame, which we leave for future work in the program analysis setting.

Contract checking [18, 13] can be used to check properties like nullness. Building on the idea of hybrid type checking [17], Xu *et al.* [25] explored how to check contracts using a hybrid of static and dynamic analysis. Their work was specialized to the context of logical assertions, whereas we are in the area of lattice-based program analyses. It is also unclear whether their approach conforms to the gradual guarantee.

O'Hearn *et al.* [20] proposed Incorrectness Logic as a means of proving that a program has a bug, rather than proving it correct. This is consistent with our goal of reducing false positives, but it stays in the realm of static reasoning, and therefore gives up soundness. In contrast, we reduce false positives without giving up soundness by adding run-time checks.

8 Conclusion

This paper is the first work on gradual program analysis. We introduced a framework which transforms abstract interpretation based static analyses relying on annotations into gradual ones. Gradual analyses handle missing annotations specially, allowing them to smoothly leverage both static and dynamic techniques. Static information is used where possible and dynamic information where necessary to reduce false positives while preserving soundness. Such analyses are also *conservative extensions* of their underlying static analyses and adhere to *gradual guarantees*, which state that losing precision is harmless. When presenting our framework, we developed a gradual null-pointer analysis, GNPA, with the previously mentioned properties that reduces false positives compared to some popular existing tools.

Importantly, the gradual framework can be applied as described to any abstract interpretation based static analysis under the following restrictions. The analysis should support annotations, have a finite-height semilattice, a monotonic, locally-sound flow function, a safety function, and operate on a first-order, procedural, imperative programming language. Additionally, checking membership in the semilattice should be decidable. Thus, initial followup work could include gradual taint analysis, to which our framework immediately applies. Finally, we do not support widening, but we do support context-sensitivity. In the future, we plan to explore extensions of our framework for infinite-height semilattices and widening; this would allow gradualization of other analyses, such as interval analysis. Still further work could include, for instance, pointer analyses, which do not have analogues in the field of gradual typing.

On the empirical side, there are further research questions to be answered: How often does a gradual analysis catch bugs statically versus how often does it catch them at run time? Is performance lost or gained when run time checks are inserted earlier via annotations rather than just-in-time? Finally, a gradual analysis will still report false positives anywhere its base static analysis is utilized and reports false positives. As a result, we plan to explore the aforementioned research questions, including the trade-off between gradual analyses reducing false positives and being conservative extensions of underlying static analyses.

References

1 Nathaniel Ayewah and William Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 241–252, 2010.

2 Johannes Bader, Jonathan Aldrich, and Éric Tanter. Gradual program verification. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 25–46. Springer, 2018.

3 Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. Nullaway: Practical type-based null safety for java. *arXiv preprint arXiv:1907.02127*, 2019.

4 Mike Barnett, Manuel Fahndrich, Francesco Logozzo, and Diego Garbervetsky. Annotations for (more) precise points-to analysis. In *IWACO 2007*, January 2007. URL: `https://www.microsoft.com/en-us/research/publication/annotations-for-more-precise-points-to-analysis/`.

5 Dan Brotherston, Werner Dietl, and Ondřej Lhoták. Granullar: Gradual nullable types for java. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, pages 87–97, New York, NY, USA, 2017. ACM. `doi:10.1145/3033019.3033032`.

6 Patrice Chalin and Perry R James. Non-null references by default in java: Alleviating the nullity annotation burden. In *European Conference on Object-Oriented Programming*, pages 227–247. Springer, 2007.

7 Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, page 258–269, New York, NY, USA, 2002. Association for Computing Machinery. `doi:10.1145/512529.512560`.

8 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL 77)*, pages 238–252, Los Angeles, CA, USA, January 1977. ACM.

9 Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

10 Sam Estep, Jenna Wise, Jonathan Aldrich, Éric Tanter, Johannes Bader, and Joshua Sunshine. Gradual program analysis for null pointers, 2021. `arXiv:2105.06081`.

11 Facebook. Infer: A tool to detect bugs in java and c/c++/objective-c code before it ships. `https://fbinfer.com/`, 2019. Accessed: 2019-10-28.

12 Facebook. Eradicate. `https://fbinfer.com/docs/checker-eradicate`, 2020. Accessed: 2021-1-10.

13 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN Conference on Functional Programming (ICFP 2002)*, pages 48–59, Pittsburgh, PA, USA, 2002. ACM.

14 Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 429–442, New York, NY, USA, 2016. ACM. `doi:10.1145/2837614.2837670`.

15 Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013.

16 Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

17 Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(2):1–34, 2010.

18 Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

19 Abel Nieto, Marianna Rapoport, Gregor Richards, and Ondřej Lhoták. Blame for null. In *European Conference on Object-Oriented Programming*, 2020.

20 Peter W O'Hearn. Incorrectness logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019.

21 Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212, 2008.

22 Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

23 Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

24 Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. Gradual verification of recursive heap data structures. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

25 Dana N Xu. Hybrid contract checking via symbolic simplification. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, pages 107–116, 2012.

Covariant Conversions (CoCo): A Design Pattern for Type-Safe Modular Software Evolution in Object-Oriented Systems

Jan Bessai
Technische Universität Dortmund, Germany

George T. Heineman
Worcester Polytechnic Institute, MA, USA

Boris Düdder
University of Copenhagen, Denmark

Abstract

Software evolution is an essential challenge for all software engineers, typically addressed solely using code versioning systems and language-specific code analysis tools. Most versioning systems view the evolution of a system as a directed acyclic graph of steps, with independent branches that could be merged. What these systems fail to provide is the ability to ensure stable APIs or that each subsequent evolution represents a cohesive extension yielding a valid system. Modular software evolution ensures that APIs remain stable, which is achieved by ensuring that only additional methods, fields, and data types are added, while treating existing modules through blackbox interfaces. Even with these restrictions, it must be possible to add new variations, fields, and methods without extensive duplication of prior module code. In contrast to most literature, our focus is on ensuring modular software evolution using mainstream object-oriented programming languages, instead of resorting to novel language extensions. We present a novel CoCo design pattern that supports type-safe covariantly overridden convert methods to transform earlier data type instances into their newest evolutionary representation to access operations that had been added later. CoCo supports both binary methods and producer methods. We validate and contrast our approach using a well-known compiler construction case study that other researchers have also investigated for modular evolution. Our resulting implementation relies on less boilerplate code, is completely type-safe, and allows clients to use normal object-oriented calling conventions. We also compare CoCo with existing approaches to the Expression Problem. We conclude by discussing how CoCo could change the direction of currently proposed Java language extensions to support closed-world assumptions about data types, as borrowed from functional programming.

2012 ACM Subject Classification Software and its engineering → Software evolution; Software and its engineering → Design patterns; Software and its engineering → Abstraction, modeling and modularity

Keywords and phrases Expression problem, software evolution, type safety, producer method, binary method

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.4

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.4`
Software: `http://doi.org/10.5281/zenodo.4756838` [2]

Acknowledgements We would like to thank the reviewers of earlier versions of this paper for their carefully thought out, detailed reviews, as well as the many constructive remarks. They helped to improve our presentation, the artifacts, and the pattern drastically. Special thanks go to the reviewer who suggested to mitigate "parameterization boilerplate" with type members.

1 Introduction

This paper presents a novel solution to the well-known Expression Problem (EP) [29], a research problem common to the fields of programming language design, multi-dimensional product line design, and software engineering. EP offers a concise representation of the challenge in implementing a system that evolves over time. The goal is to enable additive modular software evolution for data types and their methods.
The research community has identified a number of mandatory qualities that any approach must satisfy [27, 31]:

- It must be possible to add new variations, attributes, and methods to data types without changing existing software modules.
- Evolved modules must not duplicate the code of prior modules, so potential errors can be found and fixed locally.
- Hierarchical grouping of data types is an important feature of object-oriented programming that must remain intact. When a method implementation is the same for a subset of the hierarchy, it must not be necessary to repeat its definition.
- Modules must be able to evolve concurrently in branches to reflect the fact that independent features can be developed independently, possibly by different developers. It must be possible to merge these branches later so that work does not need to be duplicated.

Large systems must be able to deal with evolving dependencies without the need to rewrite existing code. This justifies the initial requirement of additive evolution, where data types and methods are considered incrementally without breaking APIs by refactoring, for example, by removing or renaming their components. Even as the software system evolves, developers must still be able to rely on compile-time checks to ensure completeness of evolutionary steps; in particular, static type-safety shall guarantee that methods are declared for all data type variants of their domain. Missing implementations shall be reported in a human-readable way at compile-time.

Solutions that apply to existing programming languages are preferable because they may provide immediate benefit to existing systems. Language extensions and new programming languages typically require years to manifest themselves in practice. They also often require rewrites of entire projects, which risk the well-known second-system effect [15]. This is also true of solutions which require code that diverges from the idiomatic use of its programming language, for example, by creating an embedded domain-specific language. There must be no restriction on the form of methods which can be added. There must be no exclusion of binary methods (methods that take data types of the evolving domain as input) or producer methods (methods which produce instances of data types in the evolving domain).

As presented in Section 2, the CoCo design pattern:

- Enables method signatures (the API) of domain logic to be stable, even when the types used in them evolve. This is facilitated by a conversion method that grants access to evolved APIs of types. It is covariantly overridden (hence the name Covariant Conversions) during the evolution process and integrated into a pattern of factories and delayed instantiation that allows implementing it safely.
- Operates fully within the constraints described above, as validated in several case studies in Section 3.
- Supports hierarchical grouping with method-deduplication [33].
- Relies only on inheritance, interfaces with default methods, and parametric polymorphism (without type bounds), which are common features of mainstream object-oriented languages [33], such as Java, Scala, and C#.
- Allows programmers to write idiomatic code in object-oriented languages to construct objects and invoke operations with method calls [30].

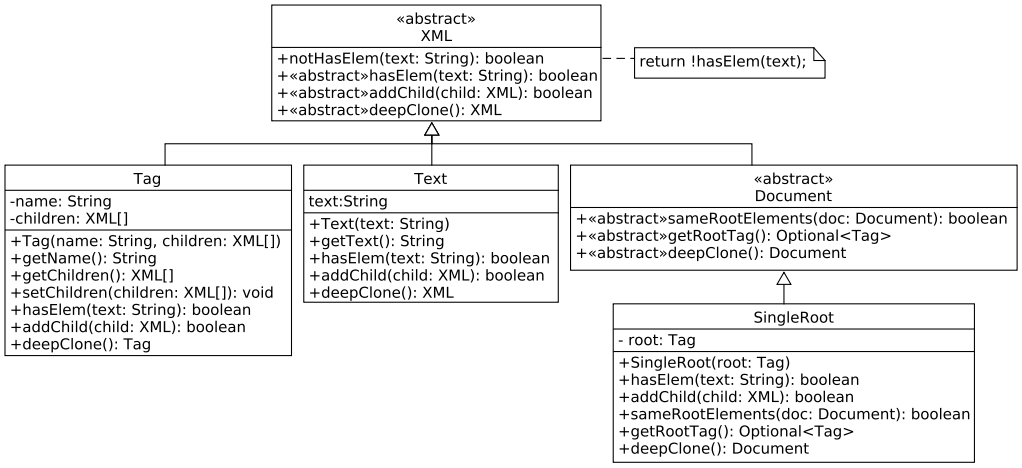


Figure 1 Example modeling XML components with multiple hierarchical classes, binary methods addChild, setChildren, sameRootElements and producer methods deepClone, getChildren, and getRootTag.

- Enables mergeable evolutionary branches [31], where developer teams can independently work on extensions that can be merged without changing or recompiling any existing code. This capability is a notable extension to EP [29] and guarantees future reusability within the code base.

Section 4 discusses how the CoCo design pattern takes a unique position in the known design space of possible solutions, where related approaches do not provide a solution to all extended constraints of EP as described above. Section 5 concludes with some remarks on pathways to broader adoption of CoCo.

2 Design Pattern

Let us consider a basic class hierarchy shown in Figure 1 to see why the CoCo design pattern is useful and how it would be applied. Assume that we want to design some classes to model XML data. We restrict ourselves to a subset of the possibilities of XML, which is interesting for demonstrating the various EP aspects discussed earlier. Figure 2 illustrates the intended use of the classes in Figure 1 with a simple XML document and its representation by a newly constructed object tree. An abstract base class **XML** is extended by classes **Tag** and **Text** as well as another abstract subdomain base class **Document**, which has a single subclass **SingleRoot**. The **XML** base class defines operations that all XML elements support. Method hasElem searches for a desired text in an **XML** element and its children. Calling hasElem on the example document from Figure 2 would return `true` for arguments `"CoCo"`, and `"relatedPattern"`, but `false` for `"Visitor"` or `"related"`. As a convenience, the default-implemented method notHasElem confirms that the desired text is not present.

Method deepClone recursively copies an entire **XML** element into a new object structure. The addChild method tries to add a given element to become a child node of an **XML** element. This might not always work and so addChild returns a boolean to indicate success. The simplest element, represented by class **Text**, represents plain text in a document. Therefore its addChild implementation does nothing and returns `false`. Objects of class **Tag** have a name and an array of children. Their addChild method appends the given element to that array and returns `true`. In contrast to **Text**, where operations work locally, the

```
<designPattern>
  <name>CoCo</name>
  <relatedPattern>Factory</relatedPattern>
</designPattern>
```

```
new SingleRoot(
  new Tag(
    "designPattern",
      new XML[] {
        new Tag("name", new XML[] { new Text("CoCo") }),
        new Tag("relatedPattern", new XML[] { new Text("Factory") })
      }
  )
);
```

Figure 2 Example XML document (top) and Java code for its construction (bottom) based on the classes shown in Figure 1.

hasElem and deepClone methods of **Tag** recursively call the appropriate methods of the XML elements known via the children array. Class **SingleRoot** is a special case of abstract class **Document**, where the topmost element is a single object of class **Tag**. Operations from **XML** are implemented performing recursive calls to the child root. Method getRootTag of **SingleRoot** returns the single root tag injected into an **Optional** wrapper class. The intended semantics of the parent method signature in **Document** is that getRootTag can be a partial method returning an empty **Optional** value for some possible implementations. Finally, sameRootElements in **SingleRoot** checks if the name of the current root tag matches the name of the root tag from the other document. This check uses getRootTag on the argument, checking if the partial result is present before comparing with the getName result.

Note that classes **XML**, **Tag**, and **Document** occur as types of parameters in addChild, setChildren, and sameRootElements. These methods are *binary methods* [4] because they involve the object on which they are called (`this`) and their parameter is also an object of a type present in the inheritance hierarchy. Methods deepClone, getChildren, and getRootTag are *producer methods* because their result is an instance produced from a type present in the inheritance hierarchy. Constructors are producer methods, and sometimes (in **Tag** and **SingleRoot**) binary methods. The return type of deepClone is *covariantly overridden* (i.e., safely replaced by a compatible subclass) in **Tag** and **Document** to enable recursive implementations that can pass cloned elements to other binary methods.

Extending the class hierarchy at any point with a new class is easy and can be done without recompiling or modifying existing code. This is, after all, the main modularity benefit of class-oriented programming. However, inserting a new method is problematic because it has to be inserted in a class, which needs to be recompiled and will cause a recursive recompilation of all sub-classes. Even worse, if the method is abstract or requires different implementations in some sub-classes, multiple classes have to change. In scenarios where the hierarchy is part of a library developed and distributed by a third-party, this is problematic and might even be impossible when the distribution is under a closed-source license.

A flawed but informative attempt to fix the issue is shown in Figure 3. Here, new classes (prefixed with E) are naively inserted to contain a new binary method validate and a new producer method asTag. Method validate is intended to use the current element as a schema to validate the given XML tag, while asTag checks if the current XML element is a tag and returns it as such, if possible. The new classes mirror the old ones, extending each of them together with their new parent. First of all, this is impossible in languages without multi-inheritance (e.g., Java). Additionally, the interplay between the new binary method validate and the old producer methods is fundamentally broken: method validate requires an

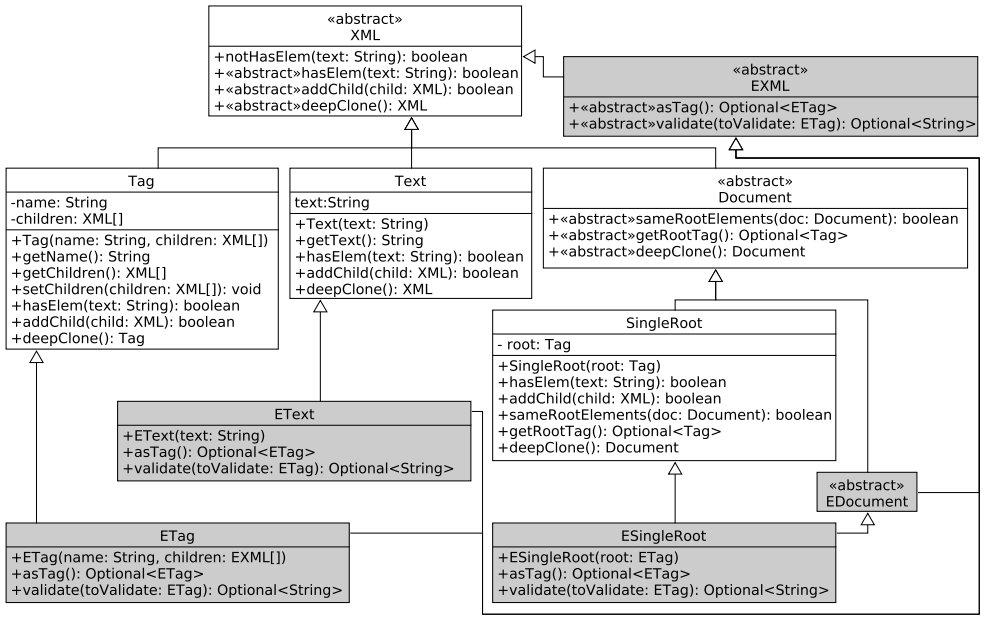


Figure 3 Flawed attempt to extend class hierarchy with a new binary method validate and a new producer method asTag provided in the extended interfaces **EXML**, **EDocument**, **ETag**, **EText**, and **ESingleRoot** (shaded background).

object of type **ETag**, while the producer methods provide objects of the old types (without an E-prefix) that were present before the extension. Wang and Oliveira [30] propose to solve the problem of multi-inheritance by turning the classes into a hierarchy of interfaces, which are implemented by some final classes that provide the code for getters, setters, and constructors. Their solution covariantly overrides all producer methods (including getters) whenever an evolution needs to add a method to the class hierarchy. In Figure 3, for example, **EXML** would become an interface with an abstract override of method deepClone(): XML to deepClone(): EXML. This trivial solution (at first glance) to the expression problem results in more problems upon closer inspection. One problem is that mutable attributes no longer work. Setters, as a special case of binary methods, cannot override their parameter type to evolved versions because method parameters are contravariant, that is, they need to be less specialized or remain the same with each inheritance step. In the example, the setChildren setter in class **Tag** would require an array of **EXML** which is more specialized than **XML**. Wang and Oliveira [30] propose to fix this by adding generic parameters with type bounds that abstract over any domain type reference. Their example remains incomplete because it does not show an extension with binary and producer methods *after these type bounds have been introduced*. In practice, the type bounds break inheritance because they covariantly modify the contravariant type of parameters of binary methods. They also break producer methods (such as deepClone) that need to construct instances of the evolved types, but invoke earlier constructors in their implementations. This forces producer code to be duplicated and modified with each evolution, violating the requirement not to duplicate code.

The CoCo design pattern eliminates the aforementioned issues by allowing references to the most generic (earliest) class type everywhere, providing abstract factory methods for constructors and conversion methods to convert earlier instances to later versions. Figure 4 shows the base class hierarchy implemented using the CoCo design pattern. Java and

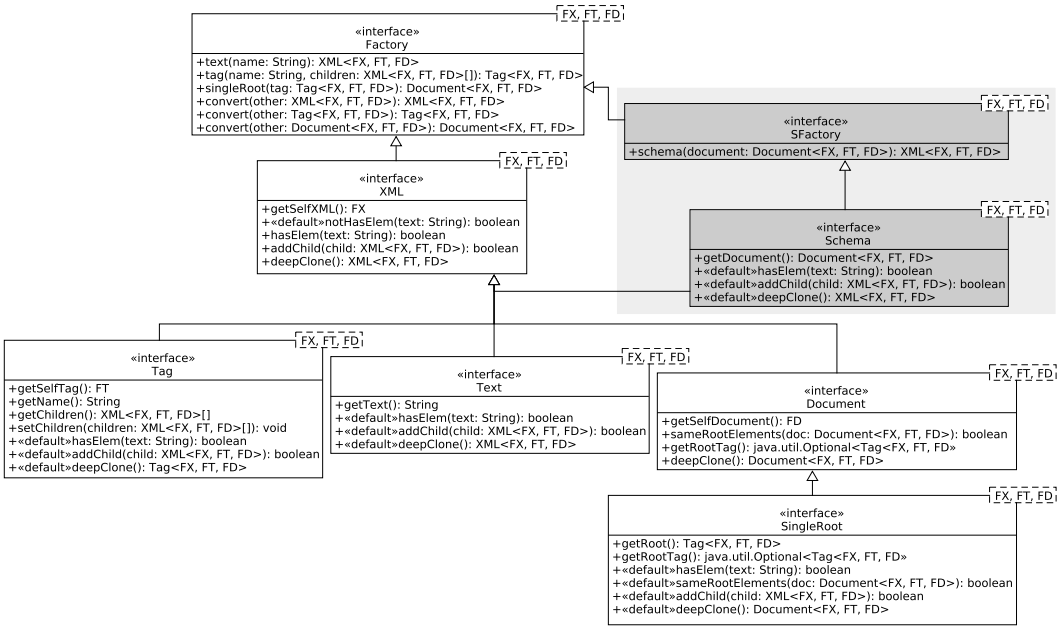


Figure 4 Hierarchy of Figure 1 implemented using the CoCo design pattern. Extension with a new data type **Schema** and factory shown in the tinted box. Methods are abstract in interfaces unless they have default implementations which are stereotyped by `«default»`.

Scala code for Figure 4 and the following figures throughout this section is available in the accompanying artifacts. As with the solution by Wang and Oliveira [30], all classes become interfaces to avoid issues with multi-inheritance. Methods are placed, as before, with implementations provided as default implementations (a feature available in Java, C#, and other mainstream OO languages). An additional **Factory** interface is introduced with abstract methods corresponding to each of the constructors of the naive object-oriented solution from Figure 1. Note that all interfaces are parameterized for domain types mentioned in the signature of methods. This parameterization allows delaying specifying which types will be finally used. A convert method is added to the factory for each parameterized type. Once an evolution is added, these convert methods will be covariantly overridden to refine their results to the latest evolutions of the converted classes. We choose to use inheritance from the **Factory** interface to make conversions and factory methods available in all parts of the type hierarchy. Conversions can be implemented in the next step, which will use the additional getSelf methods in each of the convertible interfaces. These methods are only necessary if a type is convertible (i.e., mentioned in a signature) and so the interfaces **XML**, **Document**, and **Tag** require new getSelf methods, while **Text** and **SingleRoot** can inherit them. Figure 4 also shows that the basic object-oriented feature to add new data types is not affected. The tinted box in the upper right contains a new data type **Schema**, which can be added independently of any code that was present before. Instances are instantiated by a newly-added extended Factory, **SFactory**. These additions appear in a new compilation unit without changing existing code. Unlike the Visitor pattern, CoCo does not compromise the advantage of object-oriented programming, namely, being able to freely add data types.

Figure 5 shows how the interfaces from Figure 4 are implemented by classes to allow instantiation. The implementation is fairly trivial, adding a finalized component (prefixed with F) for every component of the original class diagram. All getSelf methods are implemented by returning the current object (`this`). The convert methods of the final factories simply

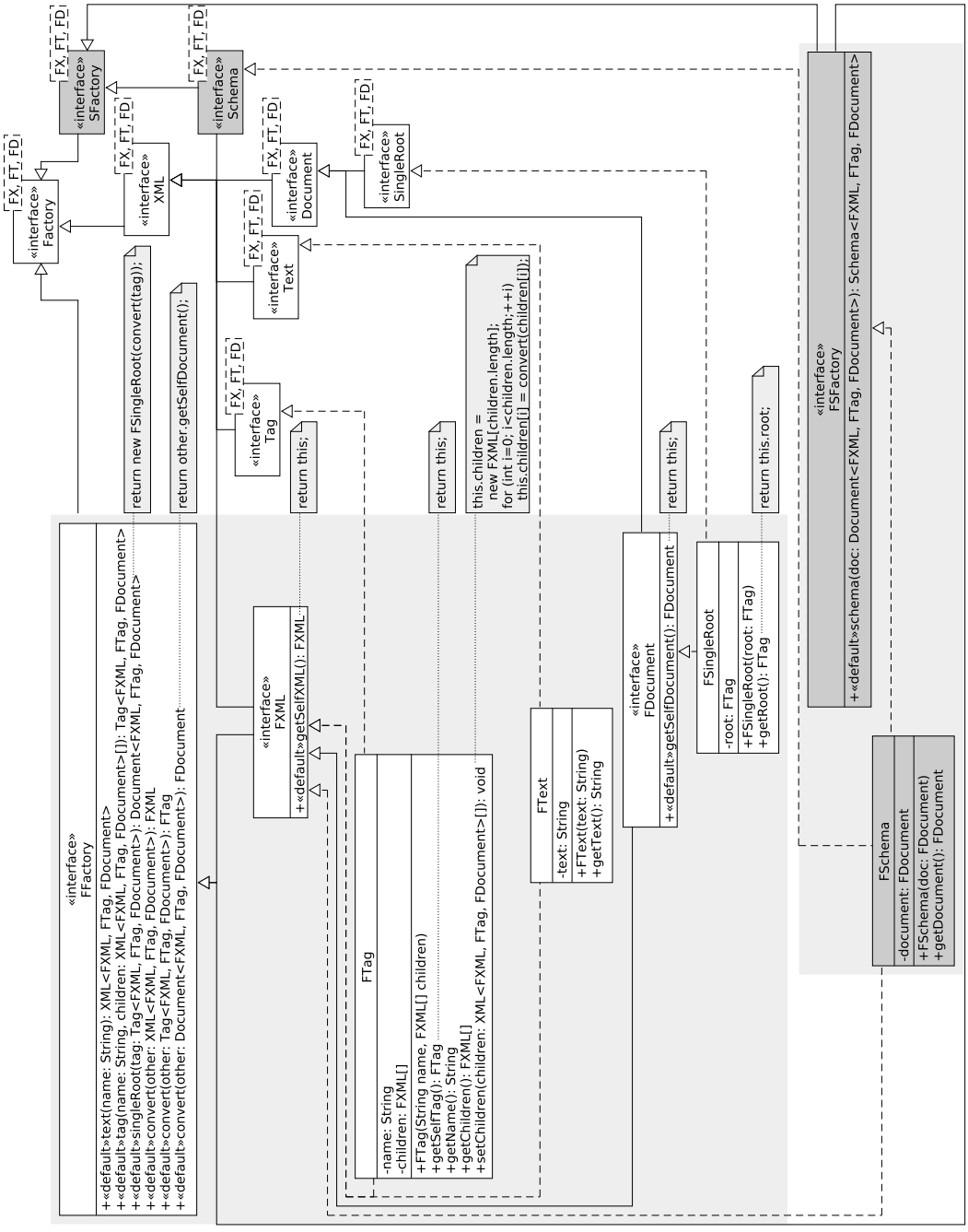


Figure 5 Final layers to instantiate the interfaces from Figure 4. The tinted upper box contains the final layer of the initial diagram, while the lower tinted box contains the separate finalized layer for the extension with **Schema**. Data types of the extension are filled in gray color. Comments in the center column show that only trivial getter, setter, constructor, and conversion code is added, which does not contain business logic or unsafe casts.

dispatch to the getSelf method of their argument. None of these methods need any casts, because in the final layer, generics parameters for the return type of getSelf are instantiated to the finalized types which inherit from the domain types at the most recent level. Delaying the implementation of convert and instantiation of generics until this moment enables convert methods (and thereby the pattern) to ensure stability of the domain logic methods, while their implementations can convert to the latest known evolution. Instantiation in factory methods is forwarded to constructor calls. Final classes (**FTag**, **FText**, **FSingleRoot**, **FSchema**) add constructors, getters, and setters with a field for each attribute reachable via the get-set-protocol prescribed by their interface. Note how the newly created layer contains no additional domain logic. In principle, a compiler extension or code generator could automatically produce it, and modern languages, such as Scala, allow to implement this layer with very few lines of code. Figure 5 already includes a (separate) final layer for the addition of the data type **Schema**.

Figure 6 shows a modular evolution that adds the binary method, validate, and the producer method, asTag, which previously failed in the naive approach. The method is added into a new extended domain interface **EXML**. Its parameter is typed by the *earliest* version of the domain interface **Tag** and does not need to evolve any further. This is possible because the implementation of validate has access to convert inherited from the extended factory interface **EFactory**, which allows to safely transform the parameter into an **ETag** instance. The *convert* methods are *covariantly* overridden with a refined result type (e.g., **EXML** instead of **XML**), which gives the CoCo design pattern its name.

Method validate is placed in the **EXML** interface and returns an optional error message if validation fails. If an element is not suitable to validate the given tag, it can return an error message. This behavior serves as a default implementation in **EXML** and is inherited in **ETag**, **EText**, **EDocument**, and **ESingleRoot**. Sharing default behavior for the general case in base classes is crucial for real world usability, which is discussed at length in [33]. Alternatively, we could have opened up **Schema** for extension by introducing it with a generic parameter, a getSelfSchema method, and a convert in **SFactory**, or we could do the same for the new type **ESchema**. In both cases validate would be available in **ESchema**, which in the first case could be obtained from any **Schema** and in the second case would serve as its own subdomain base class[1].

Implementing validate in **ESchema** is possible by recursively constructing new subschemata for the children of the document represented by the current schema. This is possible with the producer methods of the classes (for accessing children) and the factory methods from **EFactory**. Factory-produced instances are usable at the current level, because they can again be converted. The new producer method toTag is there to check if elements of the document are tags to recursively validate. It is defined in **EXML** and only overridden by the class **ETag**. Though not required in the example, extended interfaces could also choose to override an existing method introduced by an earlier extension, or require new getters and setters from their implementing final classes. Developers can remedy bad design choices (e.g., because new methods and data types can implement operations more efficiently) or add more fields to the domain data types. Since all extensions are provided in interfaces, even mainstream object-oriented languages such as Java and C# allow merging multiple domain evolutions by using multi-inheritance. In the example, this would result in **E**-prefixed classes, each extending multiple prior versions. Extended domain data types can then supply any

[1] The interested reader may find such domain extensions in the accompanying code for the TAPL case study discussed later

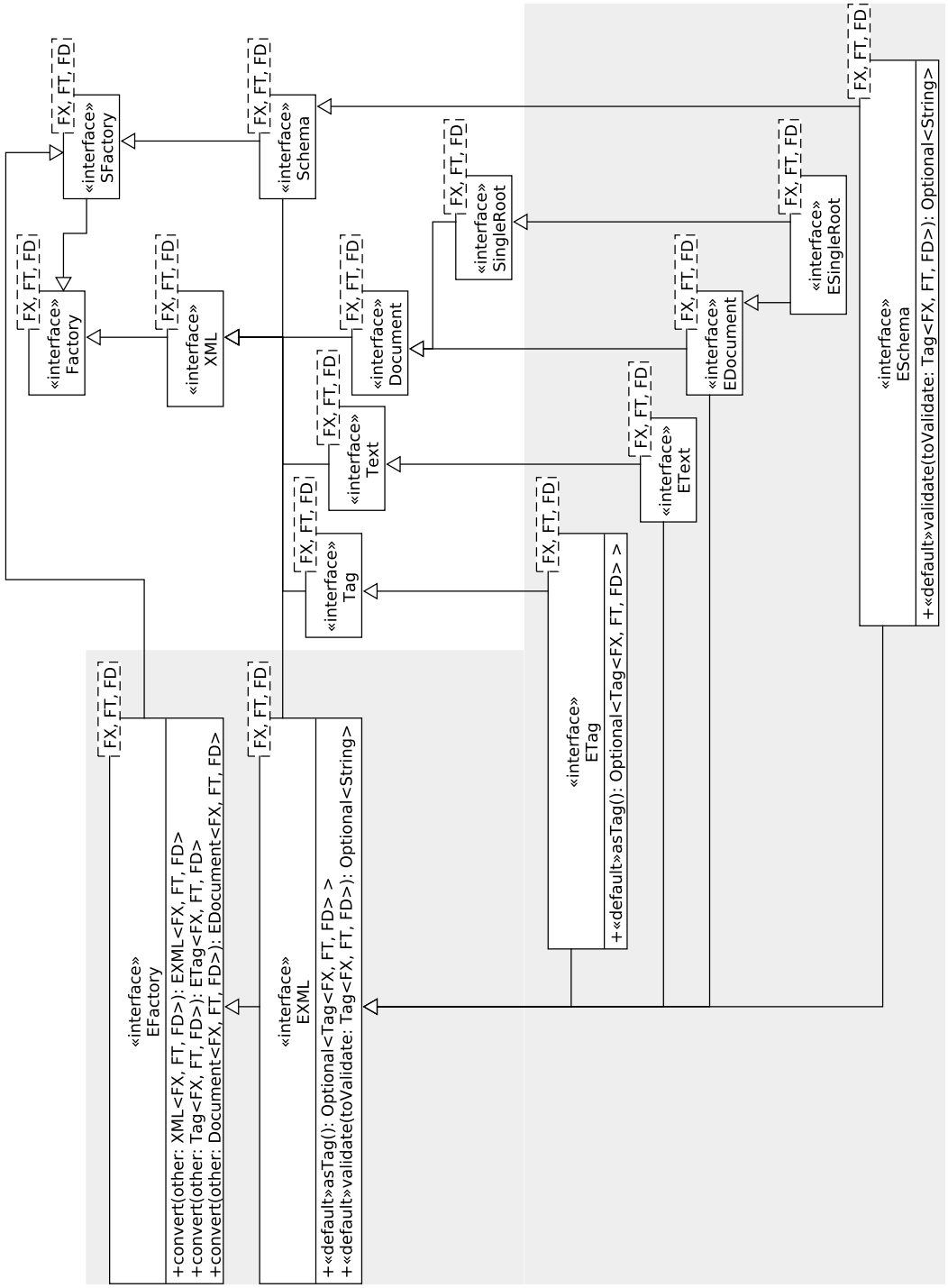


Figure 6 Extension with a new binary method toValidate and a producer method asTag. New components are placed in the tinted box. With convert, binary methods can refer to the old (less general) domain types instead of the new **E**-prefixed types, while avoiding contravariance issues from overriding parameters.

missing implementations for pairs of types and methods, where the type is present in one branch, while the method is required in interfaces from a different branch. The next section presents a case study on the traditional Expression Problem domain, illustrating why such a merge may be useful.

The class hierarchy for implementations of the extended layer from Figure 6 is shown in Figure 7. This time the extended interfaces need to be instantiated. Since extended interfaces are derived from the earlier versions, their final instantiations in **FEFactory**, **FEXML**, **FEDocument**, **FETag**, **FEText**, **FESingleRoot**, and **FESchema** remain compatible with any code that is written to work with their respective earlier versions (such as **Factory<FX,FT,FD>** or **XML<FX,FT,FD>** as long as the parameters **FX**, **FT**, **FD** remain abstract by the client. In any case, whether fixing a final version or abstractly working with its interfaces, clients can directly call methods without using visitors or object algebras. Clients also have access to convert methods via factories to ensure that objects created from producer methods provide the latest API required by the client. The final implementations are similar to those in Figure 5, but they do not contain non-trivial or domain-specific replicated code because all they do is add trivial getter, setter, constructor, and conversion methods.

```
public class Client {
 static class ClientM0<FX,FT,FD> {
  private final Factory<FX,FT,FD> factory;
  final Document<FX,FT,FD> demoDoc;
  public ClientM0(Factory<FX,FT,FD> factory) {
   this.factory = factory;
   this.demoDoc =
    factory.singleRoot(
     factory.tag("designPattern",
      factory.tag("name", factory.text("CoCo")),
      factory.tag("relatedPattern", factory.text("Factory"))));
  }
  public void run() {
   System.out.println("Has␣CoCo:␣" + demoDoc.hasElem("CoCo"));
   System.out.println("Has␣relatedPattern:␣" + demoDoc.hasElem("relatedPattern"));
   System.out.println("Has␣Visitor:␣" + demoDoc.hasElem("Visitor"));
   System.out.println("Has␣related:␣" + demoDoc.hasElem("related"));
  }
 }
 static class ClientM2<FX,FT,FD> {
  private final EFactory<FX,FT,FD> factory;
  private final ClientM0<FX,FT,FD> collaborator;
  final XML<FX,FT,FD> schema;
  public ClientM2(EFactory<FX,FT,FD> factory, ClientM0<FX,FT,FD> collaborator) {
   this.factory = factory;
   this.collaborator = collaborator;
   this.schema =
    factory.schema(factory.singleRoot(
     factory.tag("designPattern",
      factory.tag("name"),
      factory.tag("relatedPattern"))));
  }
  public void run() {
   collaborator.run();
   Optional<Tag<FX,FT,FD>> root = collaborator.demoDoc.getRootTag();
   if (root.isEmpty()) { return; }
   Optional<String> isValid = factory.convert(schema).validate(root.get());
   System.out.println("Errors:␣" + isValid.toString());
  }
 }
 public static void main(String[] args) {
  EFactory<FEXML, FETag, FEDocument> factory = new FEFactory() {};
  ClientM0<FEXML, FETag, FEDocument> client = new ClientM0<>(factory);
  ClientM2<FEXML, FETag, FEDocument> evolved = new ClientM2<>(factory, client);
  evolved.run();
 }
}
```

Listing 1 Stand alone example for evolving client code using Factories

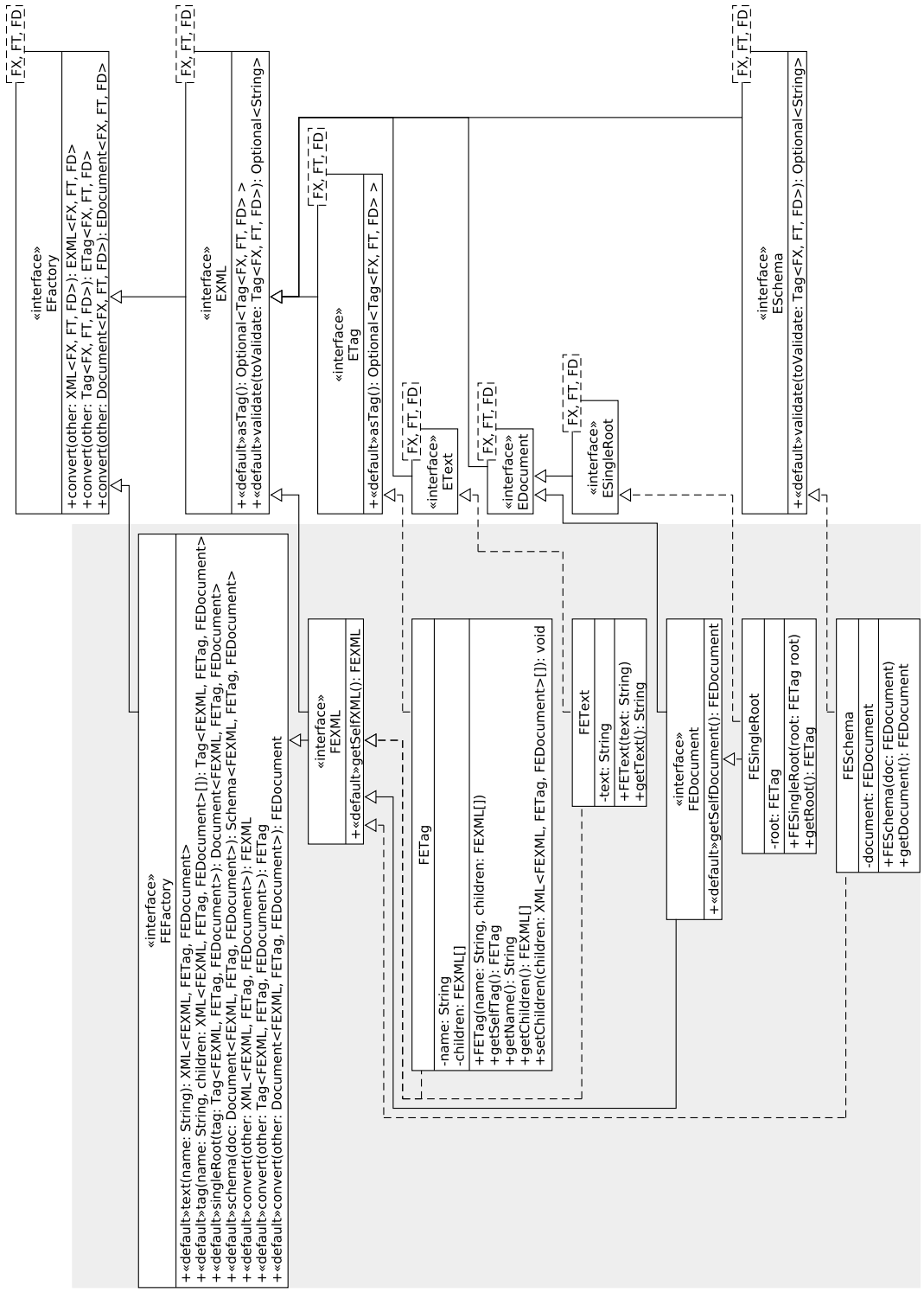


Figure 7 Final layer to instantiate the interfaces from Figure 6. New components are placed in the tinted box on the left. The finalized classes are similar to those from Figure 5, but implement the extended interfaces and instantiate generics to the newest finalized versions.

Listing 1 contains evolving client code using the pattern: the two client classes are **ClientM0** and **ClientM2** (while wrapped using a single class **Client** for this paper, this is not essential). The clients both have factories at their intended level, received as arguments via their constructor (which is just dependency injection, popular in object-oriented programming [8]). Finalized types in the clients are kept as generic parameters. The first client, **ClientM0**, operates at the first level and initializes the document from Listing 2 in its constructor. Its run method performs the previously described calls to hasElem. The second client, **ClientM2**, is passed a reference to the first, and then constructs a schema for the example XML document. Its run method interacts with the first client by calling its run method and also by using its document, and validating its root tag (if present). Only in the very last stage of the program, does the main method instantiate the generic parameters with the finalized types of the latest required level, and an **FEFactory** is passed to construct all required objects. This final construction step could also be automated by a dependency injection framework such as Guice [11]. The client code is idiomatic Java and the pattern is visible only in the passing of generic parameters and the occasional call to convert (here used to convert **Schema** to **ESchema** to gain access to its validate method). More advanced languages, such as Scala, can turn convert into an implicit conversion, automatically inserting it whenever the compiler expects a more advanced type (we will see this in the next section). In Scala we can also bundle together all generic parameters into one and use type members of path dependent types to access them[2]. This mitigates accumulation of generic parameters (sometimes called "parameterization boilerplate"). Readers with advanced Scala skills will find a demonstration of parameter bundling with path-dependent types in the artifacts for the XML example.

As we have seen in the previous example, CoCo is a design pattern rather than a framework-based approach. It is, therefore, appropriate to conclude this section using the traditional classification of design patterns established in [9].

Pattern Name and Classification

Covariant Conversions (CoCo), Creational and Behavioral Class Pattern.

Intent

CoCo structures data type classes to be extended in the future with new classes, new operations, and new fields without modifying earlier code. Type-safe convert methods transform earlier data type instances into their newest evolutionary representation to access operations added later.

Motivation

Traditional inheritance-based object-oriented programming languages do not allow methods to be added to a class hierarchy without modifying previously written code. This is known as the Expression Problem [29] and imposes particular difficulties when producer or binary methods are involved.

Applicability

Use the CoCo design pattern when

- you intend to deploy your classes in compiled form and still allow future evolutions to add new operations
- you want to merge two or more independent evolutions
- you want to introduce new hierarchy levels to an existing subtype structure
- you want to override an existing operation of an existing data type

[2] This is an idea suggested by a reviewer of this paper.

Structure
There are two families of interfaces in the pattern. A *domain interface* undergoes evolutions over time, as new data types and operations are added, and even new subdomains are identified. A *factory interface* provides the API for creating objects from the growing family of data types in the domain and a parameterized convert method that is covariantly overridden to ensure access for data types to all operations defined for the current evolution stage. Each evolution defines an *extension-factory* interface to instantiate objects of that evolution and specify the signature of the conversion method, and *extension-domain* interfaces that specify operations available for the data types in that evolution, as well as their hierarchy. Each of the inheritance relationships exists to share a signature or provide a default implementation.

Collaborations
The actual instantiation of a data type object is deferred to the finalized classes. Objects can ensure API compatibility with convert methods that invoke getSelf to return an instance of the current object at its latest evolution stage. The client code simply invokes operations on the returned objects using regular object-oriented method invocations.

Consequences
This approach minimizes code duplication by ensuring the designer can place the implementation of an operation in either an extension-domain interface or the relevant logic interfaces. An important implication of this pattern is that the compiler can statically detect missing operations in the finalized classes (i.e., a logic-interface is missing a method definition). Also, because there never is a need to dynamically cast objects, there can be no run-time exception during convert. If an evolution only adds new data types (tinted box in Figure 4), there is no need to introduce data type extension interfaces for earlier data types. Each evolution can support a hierarchy to structure the data types as needed. Domain data types need to be exposed via type variable abstractions if they are ever to be used in method signatures.

Implementation All objects are accessed through a hierarchy of domain interfaces, which has a factory interface as its base, and is refined via inheritance in subsequent evolutions, as shown in Figures 4 and 6. Instantiation occurs in a thin layer of finalized interfaces and classes. The top-level domain and factory interfaces have type parameters for every domain type that is mentioned in a signature and needs conversion. Arguments for these parameters ultimately refer to some finalized interface. Subdomain extensions are useful when parts of your subdomain have methods meant only for those subdomains. Not just a matter of having uniform access to the subdomain type hierarchy, methods could be implemented in the intermediate stages of the hierarchy to be shared throughout. Producer and binary methods may refer to the earliest points of definition of domain types in their signatures and operate using factory conversion and construction methods. This avoids any variance issues during inheritance. Client code invoking a producer operation can call convert to ensure the object conforms to the latest evolution stage. It can keep the type parameters for finalized classes abstract and only rely on non-finalized domain factories and interfaces to remain compatible with future updates to the code.

Related Patterns
CoCo uses abstract factories [9] for uniform access to object construction; as discussed in [27]; this allows for future extension by allowing instances of newer evolutions to be supplied to existing code. Accessing APIs through interfaces provided by factories is compatible with the principles of inversion of control, also known as dependency injection. Covariant overrides and finalized classes are inspired by Wang and Oliveira [30].

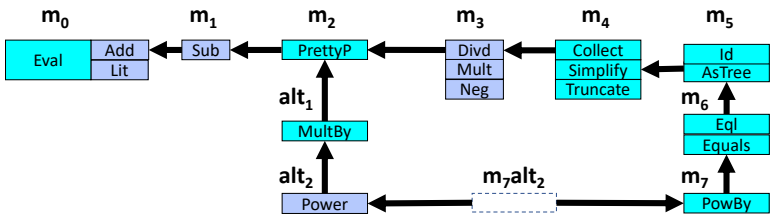


Figure 8 Extension Graph history for mathematical expressions domain.

```
package exp
trait Exp[T] extends Factory[T] { def getSelf:T }
trait Factory[T] {
 implicit def convert(other:Exp[T]) : Exp[T]
}
//////////////////////////////////////////////////////
package exp.m0
trait Exp[T] extends exp.Exp[T] with Factory[T] { def eval : Double }
trait Factory[T] extends exp.Factory[T] {
 def lit(value : Double) : exp.Exp[T]
 def add(left : exp.Exp[T], right : exp.Exp[T]) : exp.Exp[T]
 implicit override def convert(e : exp.Exp[T]) : Exp[T]
}
trait Lit[T] extends Exp[T] {
 def value : Double
 def eval : Double = value
}
trait Add[T] extends Exp[T] {
 def left : exp.Exp[T]
 def right : exp.Exp[T]
 def eval : Double = left.eval + right.eval
}

object finalized {
 trait Exp extends exp.m0.Exp[Exp] with Factory { def getSelf : Exp = this }
 trait Factory extends exp.m0.Factory[Exp] {
  override def lit(value : Double) : Exp = new Lit(value)
  override def add(left : exp.Exp[Exp], right : exp.Exp[Exp]) : Exp =
      new Add(left, right)
  override implicit def convert(e : exp.Exp[Exp]) : Exp = e.getSelf
 }

 class Lit(val value: Double) extends Exp with exp.m0.Lit[Exp]
 class Add(val left : exp.Exp[Exp],
           val right : exp.Exp[Exp]) extends Exp with exp.m0.Add[Exp]
}
```

Listing 2 Scala implementation of initial version of hierarchy

```
package exp.m1
trait Factory[T] extends exp.m0.Factory[T] {
 def sub(left : exp.Exp[T], right : exp.Exp[T]) : exp.Exp[T]
}
trait Sub[T] extends exp.m0.Exp[T] with Factory[T] {
 def left : exp.Exp[T]
 def right : exp.Exp[T]
 def eval : Double = left.eval - right.eval
}
object finalized {
 import exp.m0.finalized.Exp
 trait Factory extends exp.m1.Factory[Exp] with exp.m0.finalized.Factory {
   override def sub(left : exp.Exp[Exp], right : exp.Exp[Exp]) : Exp =
      new Sub(left, right)
 }
 class Sub(val left : exp.Exp[Exp],
           val right : exp.Exp[Exp]) extends exp.m1.Sub[Exp] with Exp with Factory
}
```

Listing 3 Modular extension adding `Sub` data type with minimal extensions required to factories

3 Case Studies

We have applied the CoCo design pattern to two standard case studies (evolving mathematical expressions and an example from a course on compiler construction) to demonstrate its effectiveness, with full implementations provided as artifacts with the paper.

The evolution history for the mathematical expression domain is captured in Figure 8 using an *extension graph* [27]. CoCo was the only pattern for which we achieved no violation of any of the constraints imposed by the Expression Problem, as we discuss in Section 4. This rich example offers a rigorous benchmark to validate any proposed EP solution. From an initial system, *m0*, with **Lit** and **Add**, evolution *m1* adds the **Sub** data type, while *m2* adds the prettyp operation that creates a string representation of an expression. An independent branch, *alt1*, diverges and adds a new producer operation, multBy, and data type, **Power** is added in *alt2*. The main branch continues development, each new evolution introducing new data types and operations as specified, such as division, multiplication, negation, literal collecting and expression simplification. Truncate is an example of an operation with a side effect. The subsequent three evolutions – *m5*, *m6*, and *m7* – introduce two binary operations equals and eql for equality checks (with equals using equality on trees computed by astree and eql dispatching to its argument instead), and a producer operation powBy for exponentiation.

```
package exp.m2
trait Exp[T] extends exp.m0.Exp[T] with Factory[T] {
 def prettyp : String
}
trait Factory[T] extends exp.m1.Factory[T] {
 implicit override def convert(e : exp.Exp[T]): exp.m2.Exp[T]
}
trait Lit[T] extends exp.m0.Lit[T] with Exp[T] {
 def prettyp : String = value.toString
}
trait Add[T] extends exp.m0.Add[T] with Exp[T] {
 def prettyp:String = String.format("(%s+%s)", left.prettyp, right.prettyp)
 // The compiler implicitly rewrites this to:
 // String.format("(%s+%s)", convert(left).prettyp, convert(right).prettyp)
}
trait Sub[T] extends exp.m1.Sub[T] with Exp[T] {
 def prettyp:String = String.format("(%s-%s)", left.prettyp, right.prettyp)
}

object finalized {
 trait Exp extends exp.m2.Exp[Exp] with Factory {
  def getSelf : Exp = this
 }
 trait Factory extends exp.m2.Factory[Exp] {
  override def lit(value : Double) : Exp = new Lit(value)
  override def add(left : exp.Exp[Exp],
                   right : exp.Exp[Exp]) : Exp = new Add(left, right)
  override def sub(left : exp.Exp[Exp],
                   right : exp.Exp[Exp]) : Exp = new Sub(left, right)

  implicit override def convert(e : exp.Exp[Exp]) : Exp = e.getSelf
 }

 class Lit(val value : Double) extends Exp with exp.m2.Lit[Exp]
 class Add(val left : exp.Exp[Exp],
           val right : exp.Exp[Exp]) extends exp.m2.Add[Exp] with Exp
 class Sub(val left : exp.Exp[Exp],
           val right : exp.Exp[Exp]) extends exp.m2.Sub[Exp] with Exp
}
```

Listing 4 Modular extension that adds `prettyp` operation, requiring extensions for existing data types to contain domain logic, and extended factories to contain implicit conversion methods

A final combined branch, *m7alt2*, merges together two independent branches, *alt2* and *m7*, leading to optimizations where powBy from the main branch is reimplemented to return newly constructed elements of type **Power** from *alt2* (and similarly for multBy and **Mult**).

This case study is fully implemented in Java, Scala, and C# 8.0 with small variations, as provided in the accompanying artifacts. The C# implementation using .NET Core 3.1 illustrating applicability to languages not based on the Java virtual machine. Aside from the language-specific syntax of C#, it completely conforms to the solution we described above.

The Scala implementation of CoCo reveals that only a small amount of boilerplate code is required. Within the space limitations of this paper, we can actually show the full code for extensions up to *m2*. Listing 2 contains the initial `exp.Exp[T]` domain interface and factory meant for a future family of data types in the domain of mathematical expressions. In Scala, interfaces with default methods are represented by traits. The `exp.Exp[T]` trait contains the signature of the `getSelf` method that returns an instance of the parameterized type, `T`. The `exp.Factory[T]` trait specifies the signature of the `convert` method that converts an `other` instance into the most recent realization of the `Exp[T]` type in the domain.

The initial definition of the system, *exp.m0*, provides the `exp.m0.Exp` trait that extends the domain with a new `eval` method computing the numerical value of a data type instance. Two new data types are defined – `Lit` and `Add` – which implement `eval` in the context of values that might (in a future evolution) be further specialized. The `finalized` object acts as a namespace to group the final trait implementations. In accordance with the pattern, they are comprised of a concrete `finalized.Factory` that offers methods to instantiate and convert the known data types, as well as trivial implementations for the domain interfaces. No domain logic of the `eval` method leaks into this finalized area.

```
package exp.m4
trait Exp[T] extends exp.m2.Exp[T] with Factory[T] {
  def simplify : exp.Exp[T]
  def truncate(level:Int) : Unit
  def collect : List[Double]
}
trait Factory[T] extends exp.m3.Factory[T] {
  implicit override def convert(toConvert:exp.Exp[T]) : exp.m4.Exp[T]
}

trait BinaryExp[T] extends Exp[T] {
  var _left : exp.Exp[T]
  var _right : exp.Exp[T]
  def left : Exp[T] = _left
  def right : Exp[T] = _right

  def truncate(level:Int) : Unit = {
    if (level > 1) {
      left.truncate(level-1)
      right.truncate(level-1)
    } else {
      _left = lit(left.eval)
      _right = lit(right.eval)
    }
  }
}

trait Add[T] extends exp.m2.Add[T] with BinaryExp[T] {
 def simplify : exp.Exp[T] = {
    if (left.eval + right.eval == 0) {
      this.lit(0)
    } else if (left.eval == 0) {
      right.simplify
    } else if (right.eval == 0) {
      left.simplify
    } else {
      this.add(left.simplify, right.simplify)
    }
  }
  def collect : List[Double] = left.collect ++ right.collect
}
```

Listing 5 Partial listing from *exp.m4* containing `BinaryExp` generic implementation

Listing 3 encapsulates the first modular evolution, *exp.m1*, adding the `Sub` data type to the system. With minimal new code (analogous to the tinted part in Figures 4 and 5), the new `Factory` classes extend existing factories to provide methods to instantiate `Sub` objects.

Listing 4 encapsulates the second modular evolution, *exp.m2*, that adds a new `prettyp` operation to the system. The existing `Exp` and `Factory` traits are refined from the prior evolution without any code duplication. The new operation must be applicable to all existing data types, so new refined types are created for `Lit`, `Add`, and `Sub`. The finalized `Factory` again instantiates and converts these refined data types. In the `prettyp` implementation for types `exp.m2.Add` and `exp.m2.Sub` we see a feature of Scala at work, which declares `convert` as an implicit method inserted by the compiler when necessary. This reduces the boilerplate code but is not strictly necessary; other languages would manually invoke `convert`.

```
package exp.m7alt2
import exp.m5.{Node, Tree}
trait Exp[T] extends exp.m7.Exp[T] with exp.alt1.Exp[T] with Factory[T] {
  override def powby(other : exp.Exp[T]) : exp.Exp[T] = power(this, other)
  override def multby(other : exp.Exp[T]) : exp.Exp[T] = mult(this, other)

  def isPower(base : exp.Exp[T], exponent : exp.Exp[T]) : Boolean = false
}
trait Power[T] extends exp.alt2.Power[T] with Factory[T] with Exp[T]
  with exp.m5.BinaryExp[T] {
  def base : exp.Exp[T] = _left
  def exponent : exp.Exp[T] = _right
  def simplify : exp.Exp[T] = {
    if (exponent.eval == 0) { lit(1) }
    else if (exponent.eval == 1) { base.simplify }
    else if (base.eval == 0) { lit(0) }
    else if (base.eval == 1) { lit(1) }
    else { power(base.simplify, exponent.simplify) }
  }
  def collect: List[Double] = base.collect ++ exponent.collect
  def id: Int = 80440
  def eql(that : exp.Exp[T]) : Boolean = that.isPower(base, exponent)
  override def isPower(base : exp.Exp[T], exponent : exp.Exp[T]) : Boolean =
    base.eql(this.base) && exponent.eql(this.exponent)
}
trait Factory[T] extends exp.alt2.Factory[T] with exp.m7.Factory[T] {
  implicit override def convert(toConvert: exp.Exp[T]) : exp.m7alt2.Exp[T]
}

object finalized {
  trait Exp extends exp.m7alt2.Exp[Exp] with Factory {
    def getSelf: Exp = this
  }

  trait Factory extends exp.m7alt2.Factory[Exp] {
    override def lit(value: Double): Exp = new Lit(value)
    override def add(left: exp.Exp[Exp], right: exp.Exp[Exp]): Exp =
      new Add(left, right)
    /* ... similar methods omitted ...*/
    override def power(base: exp.Exp[Exp], exponent: exp.Exp[Exp]): Exp =
      new Power(base, exponent)

    implicit override def convert(e: exp.Exp[Exp]): Exp = e.getSelf
  }

  class Lit(val value: Double) extends exp.m7.Lit[Exp] with Exp
  class Add(var _left: exp.Exp[Exp], var _right: exp.Exp[Exp])
    extends exp.m7.Add[Exp] with Exp

  /* ... similar class definitions omitted ... */

  class Power(var _left: exp.Exp[Exp], var _right: exp.Exp[Exp])
    extends exp.m7alt2.Power[Exp] with Exp
}
```

Listing 6 Merging branches *exp.m7* and *exp.alt2*

In evolution *exp.m4*, the `truncate` operation can be generically implemented for any expression with two recursively-defined child attributes. This is accomplished with an intermediate trait, `exp.m4.BinaryExp` shown in Listing 5, that is inherited by data types, such as `Add`, and can also be further extended by subsequent evolutions.

Listing 6 for *exp.m7alt2* shows how to merge different evolved branches. Here, `multby` and `powby` are overridden at the `Exp` level of the hierarchy to always instantiate appropriate domain data types, which become available after the merge. Traits can inherit from their predecessors in the two branches using the Scala's `with` keyword. A refinement of the trait for the exponentiation data type `exp.alt2.Power` is required to supply method implementations for the operations added in the main branch after divergence. For the new main branch data types, this would have also been possible but is not necessary in the example because the only new method in the alternative branch is `multby`, which is specified in `exp.m7alt2.Exp`. The finalized classes work exactly as expected, which is why some of their code is omitted in the listing (but available in the accompanying code repository).

Binary methods [4] are challenging because they involve the object on which they are called (i.e., `this`) and their parameter is also an object of a type present in the inheritance hierarchy. Listing 6 combines two independent branches, bringing together for the first time the `Power` data type (for exponentiation) and the `eql` operation that checks whether two mathematical expressions are equal. Our solution (coded in the `Power` trait) conforms to the strong binary method equality proposed by Zenger and Odersky [31] which dispatches on the arguments.

Listing 7 shows client code for a unit test of `exp.m2.Add`. The test is structured in a reusable version `TestTemplate` that keeps the final class parameter `T` abstract and works with the factory from `exp.m2`, as well as a concrete executable version `ActualTest` which refines the abstract class to use the implementations from `exp.m2.finalized`. This way tests are able to fully reuse the abstract part and instantiate it to use evolved finalized interfaces instead.

```
package exp.m2
import org.scalatest.FunSuite

trait TestTemplate[T] extends Factory[T] with exp.m1.TestTemplate[FT] {
  val suite : FunSuite
  import suite._

  override def test() : Unit = {
    super.test()

    val expr1 = this.add(this.lit(1.0), this.lit(2.0))
    assert("(1.0+2.0)" === expr1.prettyp)

    val expr2 = this.lit(2.0)
    assert("2.0" === expr2.prettyp)

    assert("(1.0-2.0)" === this.sub(this.lit(1.0), this.lit(2.0)).prettyp)
    assert( "((1.0-2.0)+(5.0+6.0))" === this.add(this.sub(this.lit(1.0),
    this.lit(2.0)), this.add(this.lit(5.0), this.lit(6.0))).prettyp)
   }
}

class M2Test extends FunSuite { self =>

  object ActualTest extends TestTemplate[exp.m2.finalized.Exp] with
                            finalized.Factory {
    val suite: FunSuite = self
  }

  test("M2") { ActualTest.test() }
}
```

Listing 7 Test for `exp.m2.Add` in abstract and concrete version

Table 1 Observations for the TAPL case study.

	EVF	Castor	CoCo
Duplication free domain code	no	no	yes
Fully modular	no	no	yes
Feasible w/o code generator	no	no	yes
Statically typesafe	no	yes	yes
Boilerplate free client code	no	yes	yes
Code Structuring Principle	functions	functions	classes
Human written LOC	763	768	825 (+ 862[a])
Generated LOC	1892	N/A[b]	0

[a]boilerplate for finalized class layer

[b]code generated by compiler internal macros

In our second case study, we implemented, in Java, parts of the Types and Programming Languages (TAPL) textbook by Pierce [22]. The example was also chosen to show the features of the Extended Visitor Framework (EVF) [33] and Castor [34]. Our solution implements typed and untyped compiler modules for natural numbers, Booleans, floats and strings, let-bindings, function application, and lambda-calculus. Compared to EVF and Castor, there are immediate differences in the way code and files are structured: CoCo encourages object oriented-design, placing all functionality of one domain data type evolution into one compilation unit (i.e., class or interface), while the other frameworks are inherently functional with one function definition for all domain data types corresponding to one compilation unit. This switch of perspective enabled us to find multiple *modularity violations* in the solutions provided for the other frameworks. These can be traced back to the original OCaml implementation available with the textbook. A prominent example is the pretty print function, which converts abstract syntax trees of the compiler into human-readable strings. In the CoCo solution, it is a method `print()` which returns a `String` and belongs to the generic interface `Element` for syntax tree nodes. In EVF (and similarly in Castor), pretty-printing is implemented as an object algebra returning instances of the interface `IPrint`, which provides a functional closure over a context to keep track of variable names for binders. This violates modularity because most compiler modules are not concerned with – and do not even supply – syntax for variables. In CoCo, we were able to completely avoid this issue by attaching the necessary name information to the tree upon traversal – the object-oriented view lets us cleanly encapsulate state where it is needed instead of passing it around in a map. Another modularity violation in EVF and Castor is that domain code for lambda terms is *duplicated* to add type annotations to binders. In CoCo, we simply added new fields for the evolutions that require types. Tests in EVF seem to require substantial boilerplate for algebra initialization, similar to the layer of finalized classes in CoCo, but with the crucial difference that finalized classes are not replicated by library clients. In both frameworks, EVF and Castor, users have to interact with and understand generated code, which we found mentally challenging when trying to re-engineer the case study. For EVF, this was especially problematic because of its calling convention through algebraic interfaces and a lack of type-safety where classes accept visitors of previous evolutions. In Table 1 we summarize our findings, where lines of code are counted by cloc [6] on the parts implemented in all three case studies.

4 Related Work

The CoCo Design pattern is most directly related to the approach by Wang and Oliveira [30], which also uses interfaces with covariant overrides to provide multiple inheritance as well as future refinement of data type references. In contrast, there are no issues with binary or producer methods in CoCo, and we are able to support side effects with formal setter methods. In CoCo, one only covariantly overrides the `convert` method, but in [30], one has to covariantly override every single reference to domain data types.

Harrison *et al.* describe an approach that generically abstracts over the final implementation type to improve type-safety in interface-based programming and client-side APIs [12]. From the client perspective, the APIs are similar enough to project the positive results from their case study to clients using CoCo.

From the earliest investigations into the Expression Problem, the Visitor design pattern [9] was essential, whether described by Krishnamurthi *et al.* [17] or Wadler's original email [29]. It only seems natural to turn to Visitor to support newly defined operations in subsequent evolutions.

A formal type-theoretic analysis, conducted by Oliveira [5], reveals that Visitors are related to Church encodings in functional programming languages, showing that advanced type system features, such as F-bounded polymorphism (also used in [27]), are required for type-safe Visitors that remain extensible and satisfy all constraints of EP.

Further investigations on different styles of Visitor encodings [33] reveal how to externalize Visitors and combine them with ideas from Object algebras [32]. Solutions of this form have the drawback of breaking the traditional way to design and invoke object-oriented APIs, and lead to non-standard, idiosyncratic code as discussed earlier in Section 3. External Visitors [33] provide type safety not present in normal visitors but cannot be further extended without substantial code duplication of domain logic.

EP often occurs in frameworks for designing DSLs. MontiCore [13], as well as the Revisitor implementation pattern [18], rely on Visitors. They both hide runtime typechecks behind a layer of code generated from domain-specific languages that extend Java with the explicit purpose of building DSL frameworks. Chapter 2 of [25] provides an overview of modular DSL Frameworks, which could also have been used to implement the compiler case study in Section 3.

Verna [28] provides a detailed account of the practical issues that arise from implementing the Visitor design pattern. In consensus with our observations, the problems with Visitor include non-idiomatic calling conventions and lack of extensibility, without losing type-safety or forcing code duplication. CoCo eliminates all these issues by avoiding Visitors, relying on the idiomatic placement of methods in domain data type interfaces, and providing type-safe conversion methods.

The Castor framework [34], which is a follow-up to EVF [33], requires self-type annotations, path-dependent types, and traits. This combination is (to our knowledge) only available in Scala. Despite its heavy requirements, Castor does not provide a complete EP solution, because (as the authors acknowledge) nested pattern-matching is not checked for exhaustiveness. In CoCo, no similar problem occurs because dispatching on children is always safe, and default implementations are properly placed in domain data type interfaces. Zhang and Oliveira [34] observe that avoiding exhaustiveness issues is possible by either adding new language features to Scala or duplicating default logic for new data types, in violation of requirements for EP. The more pressing issue, however, is forcing programmers to rewrite existing systems in Scala to realize the benefits from Castor. Additionally, the

Scala sub-dialect used by Castor relies on advanced language features, such as macros, which are inaccessible to novice programmers and introduce difficult to understand compiler error messages.

The CoCo design pattern is immediately applicable to numerous mainstream programming languages, such as Java, C#, and C++. However, it cannot be used in Rust or a multi-paradigm language such as Go because both programming languages have discarded class-based inheritance hierarchies in favor of constructs akin to type classes from functional programming languages, such as Haskell. This switch was motivated because of prominent solutions to the EP in functional programming languages, including tagless final [16] and trees that grow [19].

Language extensions are routinely proposed, such as extensible pattern matching with extractors [26], but this introduces compatibility issues with existing code; it additionally requires code generators for substantial boilerplate. Many proposed language extensions deal with the problem of self-types, which was studied in the context of family polymorphism [7]. In essence, the idea is to *existentially quantify* over the domain type and bound the existential quantification by the domain type at the current evolution level. Evidence for this quantification is then associated with each instantiated object and a way to return the current object as an instance of the existentially quantified type is given. This is in sharp contrast to CoCo, where we *universally qualify* over the domain type and avoid any type bounds. Instead, all conversion is centralized in a convert-method and delayed until the last possible point in a finalized getSelf-method, which is no longer generic and thereby does not have to deal with type-bounds. Listing 8 shows a short snippet of Scala, which allows both encodings, to illustrate the essential difference.

```
trait Exp[T] {
  def getSelf: T                  // No bound on T
}
trait FExp extends Exp[FExp] {    // Bound ensured in finalized instance
  def getSelf: FExp = this
}
trait ExpFamily {
  type Self <: ExpFamily          // Bound to ensure compatibility
  def getSelf: Self               // Returned as a compatible type
}
```

Listing 8 Scala code illustrating CoCo vs existentially encoded family polymorphism

Saito *et al.* [24] show how to extend a minimal Java core calculus with the features necessary for family polymorphism. The idea can also be rephrased with path types [14], which is why the more powerful dependent object types [1] of Scala are so suitable to illustrate it. Still, practical integration into programming languages poses serious challenges, which are beyond the scope of this work, but are addressed with solution proposals in [23, 35]. The latter proposal [35] might be interesting for future work to reconcile CoCo with languages such as Rust because it combines family polymorphism with type classes.

The program languages research community has extensively studied EP since its initial formulation by Wadler in 1998 [29]. Wadler proposed an experimental language, GJ, based on Visitor using a language mechanism to allow a type variable to be indexed by any inner class defined in that variable's bound (similar to the bound on the inner type presented in Listing 8). This requires projections out of generic types [21] which encounters soundness issues and was not added to Java, and even partially dropped from Scala 3.0. The Extensible Visitor [17] requires a runtime check in a Java solution. The Interpreter design pattern [9] has also been suggested to solve EP, but its Factory classes would have to be modified whenever new data types are added. Object Algebras [20] similarly use interfaces to define the evolving interface of the system while factory objects provide concrete implementations. However,

Table 2 Language features necessary for the CoCo design pattern, EP approaches, and related work.

Approach	Data type/ operation extension	Producer methods	Binary methods	Merging independent evolutions	Hierarchical ordering of subdomains
CoCo	interfaces with default methods; covariant overriding of `convert` method return type; parametric polymorphism for `getSelf`	multi-inheritance from interfaces	`convert` method	multi-inheritance from interfaces	inheritance
Trivially [30]	inheritance and interfaces with covariant overriding of return types	not available without violates EP	not available without violates EP	multi-trait-inheritance in trait-based languages	not discussed
Extensible Visitor [17]	inheritance and dynamic cast (violates EP); parametric polymorphism; single-class inheritance	inheritance	method overriding	multi-inheritance from interfaces	not discussed
Interpreter [3]	inheritance and dynamic cast (violates EP)	duplicate methods (violates EP)	dynamic cast (violates EP)	multi-inheritance from interfaces	not discussed
Torgersen [27]	inheritance and dynamic cast (violates EP); final methods; parametric F-bounded polymorphism	not discussed	not discussed	multi-class inheritance	not discussed
EVF [33]	inheritance; parametric polymorphism; interfaces with default methods; multi-inheritance from interfaces; annotation-based macros; lack of type-safety for earlier visitors (violates EP)	parametric polymorphism	parametric polymorphism	multi-inheritance from interfaces	inheritance
Castor [34]	path-dependent types; self-type annotations; multi-trait-inheritance in trait-based languages; partial pattern matching on types (violates EP)	not discussed	not discussed	multi-trait-inheritance in trait-based languages	inheritance

supporting producer methods is only possible when the factory object algebras have access to "the latest" object algebra in the evolution history, which can be accomplished by modifying a special "combined" object algebra that composes together all known factories. This modifies existing code and, in addition, working with object algebras involves considerable boilerplate code for clients, to the point that researchers recommend using code generators [32]. Table 2 summarizes the language features required by various EP solutions.

5 Conclusion and Future Work

The CoCo design pattern combines a number of programming idioms commonly used in object-oriented design (abstract factories, access to and sharing of implementations through interfaces, dependency inversion) with the novel addition of *covariantly* overridden *conversion* methods. This allows the modular future extensibility of class hierarchies with new data types, methods, and fields without code duplication.

We have illustrated how this solves the Expression Problem within the constraints of mainstream object oriented languages, improves modularity, and reduces the amount of boilerplate when compared to other EP approaches. It satisfies the constraints for a "full and

final" solution as summarized by Torgersen [27]. While feasible without tool support, a path for widescale adoption of the pattern should consider compiler assistance for generating the boilerplate code required for the finalized class layer. Scala's implicit conversions are among useful compiler features to make CoCo more straightforward. However, relying on compiler extensions would require fixing a particular language and semantics which we *intentionally avoided* here, to leave the pattern applicable to a broad range of languages. In this line of future work, a precise formal definition and proofs about it become meaningful and should be provided. A further question for future work will be if the additional structure exposed by the pattern can be exploited in code analysis tools to provide better insights into the evolution and code quality of projects. One of the main contributions of the CoCo design pattern is to illustrate that the current trend toward integrating functional programming and specifically pattern matching into object oriented languages (e.g., JEP 394 [10]) is not necessarily the only way forward.

CoCo avoids unsafe instance-of pattern matching and the alternative closed-world assumption (i.e., data types can no longer be extended) to make it safe, without adding new features to the type system **and** remaining compatible with the object-oriented paradigm of programming. We also hope that CoCo solves some of the prevailing issues around the overuse of the visitor pattern [28].

References

1 Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 249–272. Springer, 2016. doi:10.1007/978-3-319-30936-1_14.

2 Jan Bessai, George Heineman, and Boris Düdder. JanBessai/ecoop2021artifacts: State published with paper, 2021. doi:10.5281/zenodo.4756838.

3 Kim B. Bruce. Some challenging typing issues in object-oriented languages. *Electron. Notes Theor. Comput. Sci.*, 82(7):1–29, 2003. doi:10.1016/S1571-0661(04)80799-0.

4 William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In Frances E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 125–135. ACM Press, 1990. doi:10.1145/96709.96721.

5 Bruno C. d. S. Oliveira. Modular visitor components. In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*, pages 269–293. Springer, 2009. doi:10.1007/978-3-642-03013-0_13.

6 Al Danial. Cloc code analysis tool, September 2020. URL: https://github.com/AlDanial/cloc.

7 Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer, 2001. doi:10.1007/3-540-45337-7_17.

8 M. Fowler. Inversion of Control Containers and the Dependency Injection pattern, 2004. URL: http://martinfowler.com/articles/injection.html.

9 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

10 Brian Goetz and Gavin Bierman. JEP 394: Pattern matching for instanceof. Technical report, Open JDK, Oracle Corporation, 2021. URL: http://openjdk.java.net/jeps/394.

11 Guice Framework for Java, 2021. URL: https://github.com/google/guice.

12 William Harrison, David Lievens, and Fabio Simeoni. Safer typing of complex API usage through Java generics. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, page 67–75, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1596655.1596666.

13 Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional language engineering using generated, extensible, static type-safe visitors. In Andrzej Wasowski and Henrik Lönn, editors, *Modelling Foundations and Applications - 12th European Conference, ECMFA@STAF 2016, Vienna, Austria, July 6-7, 2016, Proceedings*, volume 9764 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2016. doi:10.1007/978-3-319-42061-5_5.

14 Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 113–132. ACM, 2007. doi:10.1145/1297027.1297037.

15 Frederick P. Brooks Jr. *The mythical man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1995.

16 Oleg Kiselyov. Typed tagless final interpreters. In Jeremy Gibbons, editor, *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, volume 7470 of *Lecture Notes in Computer Science*, pages 130–174. Springer, 2010. doi:10.1007/978-3-642-32202-0_3.

17 Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In Eric Jul, editor, *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*, volume 1445 of *Lecture Notes in Computer Science*, pages 91–113. Springer, 1998. doi:10.1007/BFb0054088.

18 Manuel Leduc, Thomas Degueule, Benoît Combemale, Tijs van der Storm, and Olivier Barais. Revisiting visitors for modular extension of executable DSMLs. In *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017*, pages 112–122. IEEE Computer Society, 2017. doi:10.1109/MODELS.2017.23.

19 Shayan Najd and Simon Peyton Jones. Trees that grow. *J. Univers. Comput. Sci.*, 23(1):42–62, 2017. URL: http://www.jucs.org/jucs_23_1/trees_that_grow.

20 Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 2–27, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

21 Lionel Parreaux. What is type projection in Scala, and why is it unsound?, 2019. Blog Entry. URL: https://lptk.github.io/programming/2019/09/13/type-projection.html.

22 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. URL: https://www.cis.upenn.edu/~bcpierce/tapl/.

23 Sukyoung Ryu. ThisType for object-oriented languages: From theory to practice. *ACM Trans. Program. Lang. Syst.*, 38(3):8:1–8:66, 2016. doi:10.1145/2888392.

24 Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *J. Funct. Program.*, 18(3):285–331, 2008. doi:10.1017/S0956796807006405.

25 Stefan Sobernig. *Variable Domain-specific Software Languages with DjDSL - Design and Implementation*. Springer, 2020. doi:10.1007/978-3-030-42152-6.

26 Nicolas Stucki, Paolo G. Giarrusso, and Martin Odersky. Truly abstract interfaces for algebraic data types: the extractor typing problem. In Sebastian Erdweg and Bruno C. d. S. Oliveira, editors, *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala, SCALA@ICFP 2018, St. Louis, MO, USA, September 28, 2018*, pages 56–60. ACM, 2018. doi:10.1145/3241653.3241658.

27 Mads Torgersen. The Expression Problem Revisited. In Martin Odersky, editor, *Proceedings of the 18th European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 123–143. Springer International Publishing, 2004. doi:10.1007/978-3-540-24851-4_6.

28 Didier Verna. Revisiting the visitor: the "just do it" pattern. *J. Univers. Comput. Sci.*, 16(2):246–270, 2010. doi:10.3217/jucs-016-02-0246.

29 Philip Wadler. The expression problem, 1998. E-Mail to the Java Genericity Mailing List. URL: http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

30 Yanlin Wang and Bruno C. d. S. Oliveira. The expression problem, trivially! In Lidia Fuentes, Don S. Batory, and Krzysztof Czarnecki, editors, *Proceedings of the 15th International Conference on Modularity, MODULARITY 2016, Málaga, Spain, March 14 - 18, 2016*, pages 37–41. ACM, 2016. doi:10.1145/2889443.2889448.

31 Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem, 2004. URL: http://infoscience.epfl.ch/record/52625.

32 Haoyuan Zhang, Zewei Chu, Bruno C. d. S. Oliveira, and Tijs van der Storm. Scrap your boilerplate with object algebras. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 127–146. ACM, 2015. doi:10.1145/2814270.2814279.

33 Weixin Zhang and Bruno C. d. S. Oliveira. EVF: an extensible and expressive visitor framework for programming language reuse. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 29:1–29:32. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ECOOP.2017.29.

34 Weixin Zhang and Bruno C. d. S. Oliveira. CASTOR: Programming with extensible generative visitors. *Sci. Comput. Program.*, 193:102449, 2020. doi:10.1016/j.scico.2020.102449.

35 Yizhou Zhang and Andrew C. Myers. Familia: unifying interfaces, type classes, and family polymorphism. *Proc. ACM Program. Lang.*, 1(OOPSLA):70:1–70:31, 2017. doi:10.1145/3133894.

ALPACAS: A Language for Parametric Assessment of Critical Architecture Safety

Maxime Buyse
Uber Elevate, Paris, France

Rémi Delmas
Uber Elevate, Paris, France

Youssef Hamadi
Uber Elevate, Paris, France

Abstract

This paper introduces ALPACAS, a domain-specific language and algorithms aimed at architecture modeling and safety assessment for critical systems. It allows to study the effects of random and systematic faults on complex critical systems and their reliability. The underlying semantic framework of the language is Stochastic Guarded Transition Systems, for which ALPACAS provides a feature-rich declarative modeling language and algorithms for symbolic analysis and Monte-Carlo simulation, allowing to compute safety indicators such as minimal cutsets and reliability. Built as a domain-specific language deeply embedded in Scala 3, ALPACAS offers generic modeling capabilities and type-safety unparalleled in other existing safety assessment frameworks. This improved expressive power allows to address complex system modeling tasks, such as formalizing the architectural design space of a critical function, and exploring it to identify the most reliable variant. The features and algorithms of ALPACAS are illustrated on a case study of a thrust allocation and power dispatch system for an electric vertical takeoff and landing aircraft.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Computer systems organization → Embedded and cyber-physical systems

Keywords and phrases Domain-Specific Language, Deep Embedding, Scala 3, Architecture Modelling, Safety Assessment, Static Analysis, Monte-Carlo Methods

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.5

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.14`

1 Introduction

The work presented in this paper is motivated by the emergence of Urban Air Mobility (UAM) which will move people and cargo by air, exploiting the third dimension to escape ground congestion. UAM will be powered by new electric Vertical Take-Off and Landing (eVTOL) aircraft. They will use highly redundant fully electric propulsion systems for reduced noise and safe operation in urban areas. The Aerospace Recommended Practices (ARP-4754A[1]/4761[2]) guide the design and certification process of these aircraft. According to [18], safety assessment is very challenging for eVTOL development with large costs associated to safety modeling, and difficulties to assess and optimize multiple architecture variants.

New eVTOL companies propose very different system architectures (lift-only configurations, lift+cruise configurations with tilt-wing, tilt-rotor, etc.) for a wide variety of applications (air taxi, deliveries, freight, etc.) and safety aspects play a decisive role in the

[1] `https://www.sae.org/standards/content/arp4754a/`
[2] `https://www.sae.org/standards/content/arp4761/`

competition of designs. Moreover, exploring the underlying design-space from a safety and certification perspective can help define meaningful mandatory safety targets, which are still being actively discussed by regulators in the US and the EU.

A system is called *critical* when the failure to perform its function is likely to result in loss of life or extreme environment damage. Examples of critical systems are embedded aircraft control systems, railway control systems, nuclear plant control systems, radiotherapy equipment control systems, etc. The acceptable risk levels for critical systems are defined by competent regulatory bodies, collaboratively with stakeholders such as system providers, system users, the state, etc. The severity of identified risks determines fault tolerance and reliability requirements for the system, as well as design and verification process requirements. *System safety assessment* consists in characterizing the risk for a particular system, identifying applicable safety requirements and demonstrating that the planned system architecture meets safety requirements.

All phases of the safety process are backed by modeling and analysis tasks in some adapted formalism. The modeling artifacts are used as evidence in the certification process. Implementation requirements [11] are derived from the safety analysis to feed the implementation phase following DO-178C and DO-254A recommendations. Similar concepts apply in other domains such as automotive and railway [39] [46].

The safety, verification and validation activities of critical embedded systems account for a large part of the total development cost. Identifying the optimal system architecture according to safety metrics and implementation cost criteria before starting its implementation and certification is hence essential, in particular in the UAM domain where designs are created from a blank slate without preexisting reference or safety record. A lack of agility in these early design phases can result in suboptimal system designs and limit programmatic agility in the long run, i.e. the ability to update an existing system with new functions or safety enhancing features that would require substantial modifications of the safety models and analysis.

As will be seen in the related works section, current safety formalisms lack features which could make safety modeling more efficient. These features are commonly found in modern functional and object-oriented programming languages: encapsulation, generic parameters, higher-order parameters, polymorphism, etc. Better support for incremental and generic modeling can allow to go beyond safety assessment and support genuine safety-driven *design-space exploration*, where optimal design decisions are made by comparing automatically several candidate system architectures. For this we propose the Alpacas safety formalism, built as an embedded domain-specific language in the Scala 3 functional programming language. Alpacas offers first-class generic and parametric modeling capabilities allowing to formalize higher-order design spaces. The embedding allows to fuse declarative safety modeling and programming in a coherent framework, to compute safety indicators for system variants more easily, effectively unlocking architectural design-space exploration and optimization.

The rest of the paper is structured as follows: Section 2 reviews existing safety modeling formalisms and their limitations, as well as domain-specific language implementation techniques; Section 3 presents the design goals and requirements that shaped Alpacas, together with a running example; Section 4 introduces the Alpacas syntax and implementation using the running example; Section 5 describes the formal semantics of Alpacas; Section 6 discusses safety analysis algorithms provided by Alpacas; Section 7 describes a design-space exploration study performed with Alpacas for a thrust reallocation function of an electric vertical takeoff and landing aircraft; Last, Section 8 concludes the paper and outlines perspectives to this work.

2 Related works

We present core safety modeling concepts in Section 2.1, related works on safety modeling and analysis in Section 2.2, as well as relevant literature on domain-specific language implementation techniques in Section 2.3.

2.1 Core safety concepts

We now review fundamental concepts in system safety modeling as originally presented in [47]. A *System* is an assembly of *Components*, operating together to perform a *Function*. *Basic Failure Events* cause changes of the internal *State* of components. At the very least, a component has two states: *working* and *failed*, but it can have more, such as multiple functional or degraded modes. *Failure Modes* are the external manifestations of the internal failure state of a component. For instance, a valve component could be in three states: *working*, *stuck-open*, *stuck-closed*, with corresponding failure modes *nominal pressure*, *over-pressure* or *under-pressure*, respectively. Failure modes propagate and combine through the system, affecting its ability to perform its function. A *Failure Condition* is a failure mode of the function performed by the system, and it is the consequence of one or more basic failure events. The *Structure Function* of the system specifies how basic failure event combinations or sequences produce different failure conditions at the system level.

Failure events occur randomly following certain delay distributions, failure behaviour can be non-monotonic and sensitive to event ordering, propagations can exhibit some level of randomness and time dependency, which makes safety modeling and analysis a complex problem. Many formalisms have been proposed, depending on the class of system to analyze. In all cases, safety models are built in order to compute *safety indicators* of a system and predict its performance. *Qualitative indicators* describe the logical relationship between basic failure events and system failure conditions. *Minimal cutsets or sequences* (MCS) are minimal event combinations or sequences triggering a failure condition. *Quantitative indicators* capture the probabilistic aspects of system failure. For instance, *Unreliability*, the probability that the system fails in the interval $[0, T]$, depends in a non-trivial way on basic event probabilities and on system architecture.

2.2 Safety formalisms

Safety formalisms are distinguished by their semantics, which delimits the class of real-world systems they can faithfully model. Semantics also influences the tractability of safety indicators. The other major aspect for use in real-world applications is the level of support for design-space exploration, i.e. the ease with which models can be parameterized, updated, extended, reused, etc. Each new system design iteration alters the system architecture and its dysfunctional behaviour, which must be reflected in the safety model. Design modifications are also largely guided by the safety analysis of different design options which orient the choice of fault-tolerance patterns, redundancy levels, basic event occurrence rates, etc.

The most widely used safety formalism in industrial domains are *Fault Trees* [30] and *Bow-Tie Diagrams* [22]. These graphical formalisms address *static systems* where the order of event occurrences does not matter, and allow a direct representation of combinatorial structure functions as Boolean functions over basic events interpreted as propositions. Dynamic fault trees [24] extend fault trees to handle *dynamic systems* where event ordering matters, by adding logic gates where subtree ordering encodes temporal sequencing constraints. Dynamic systems are also traditionally modeled using Markov chains. In *non-repairable systems*, new

events can only degrade the health of the system, which translates to monotony properties of structure functions. In *repairable* systems, a new event can improve the health of the system by prompting a repair action. Boolean logic-driven Markov Processes [13, 16, 32, 33] allow to address dynamic repairable systems.

Model-checking tools such as PRISM [35, 36] or UPPAAL-SMC [19, 17], supporting formalisms like Continuous-time Markov chains (CTMC) or Probabilistic Timed Automata (PTA), can be used for reliability analysis. Generalized Semi-Markov Processes (GSMP), which are strictly more expressive than CTMC and PTA, have also been quite successful for reliability analysis using Monte-Carlo [37, 23, 48] or bounded model-checking approaches [2]. Works such as [25] propose a superset of both GSMPs and PTAs and leverage either Monte-Carlo simulation or PRISM as back-end depending on the particular subset the model falls in.

In all of the above formalisms, system architecture, components, failure modes and failure propagation are not first class concepts, the concept of failure condition is implicit and cannot be disentangled from models and models are not composable. Moreover, design-space formalization is impossible with these formalisms, for their lack of generic modeling features and inability to express parametric system families.

The more recent *Model-Based Safety Analysis* (MBSA) approach [38] addresses these issues by adopting hierarchical modeling, failure modes, propagation rules and failure conditions as first-class concepts. A first collection of works proposes to annotate a functional design model with failure mode propagation rules: [20] proposes a safety extension for the well known AADL system design language; [31] extends a Simulink model with Boolean formulas modeling failure mode propagation conditions; in xSAP [12] a reference functional model is annotated with timed failure propagation information.

Extending a functional model with safety information is debatable, due to the fact that fault propagation can occur through non-functional paths in real systems, and that external non-functional factors also need to be modeled to conduct safety assessment. The computation of safety indicators requires to abstract away safety-irrelevant aspects of system behaviour to become tractable, and results in models that are qualitatively different from engineering models. Another line of works in MBSA addresses these issues by proposing languages dedicated to safety modeling. In particular, the Altarica family of languages [4, 42, 9] proposes a hierarchical modeling approach based on components and data-flow with a semantics based on Stochastic Guarded Transition Systems (SGTS). This framework is at least as expressive as GSMP and allows to model dynamic and repairable systems, with concurrency and real-time aspects, with deterministic or stochastic failure mode propagation rules, common-cause failure modeling with event synchronizations. The recent S2ML framework [8] uses concepts borrowed from object-oriented programming to improve model reuse and allow the creation of component libraries, and only offers a restricted form of parametricity.

Alpacas is a new incarnation of SGTS with hierarchical modeling and expressivity comparable to Altarica. However, Alpacas is tailored for design-space exploration by adding first-class support for generic modeling based on functional programming concepts such as higher-order parameters, typeclass polymorphism, etc. Design-space formalization, was only handled externally and informally in all previous approaches. In addition, Alpacas removes the strict boundary between safety models and analysis algorithms, opening the way to better design-space exploration methods.

2.3 Domain-specific languages

Domain-Specific Languages (DSL) are dedicated to the modeling and solving of particular classes of problems, and are generally not complete programming languages. *Standalone DSLs* are implemented by writing a standalone front-end (lexer, parser, type-checker, ...) and back-end (interpreter, compiler, solver, optimizer, ...). *Embedded DSLs* on the other hand are implemented within a host language [28], and exposed to the user through functional combinators or syntax extensions. Language embedding allows to reuse the host language syntax, type system, semantics, libraries, compilers and tools at the cost of slightly less freedom in the syntax definition of the DSL, and has become a very popular approach. A DSL embedding is *shallow* when DSL constructs are directly interpreted in the host language without any further analysis or code generation stages. The *Tagless Final* approach [34] is very popular for shallow DSL implementation: DSL operations are represented as a purely functional interface parameterized by a monadic higher-kinded effect type, which defines its semantics. In *deep embedding* approaches, evaluating the domain-specific program yields a term data structure representing the DSL program that is then analyzed and processed in multiple stages [44]. Deep embedding approaches based on free monads have been proposed, however both shallow and deeply embedded monadic approaches are hard to scale to large DSLs, are syntactically constrained by the monadic programming style, and require deep understanding of monads and higher-kinded types from the end user.

To implement Alpacas, we opted for a non-monadic deep embedding technique, because the language is relatively rich and requires advanced static checks and preprocessing on the models before running simulations and analyses. The Scala language is known to offer very good support for deep embedding and staging, as demonstrated in multiple domains like hardware description with the Chisel language [5], Lightweight Multi-Stage numerical code optimization [44], full language virtualization [43], GPU acceleration of numerical code [49], event monitoring with automata [26], polymorphic linear algebra [45], etc. The newly released Scala 3 based on the Dependent Object Type calculus [3] offers even better support for deep embedding, with generalized algebraic data types, extension methods, infix methods, contextual abstraction mechanisms such as type-classes and automatic type-class derivation, and more importantly implicit function types [40], etc. Support for Multi-stage programming is also improved with the new inline-def macro system which, together with a new quoting and splicing system, provides efficient compile-time as well as run-time code generation.

3 Generic modeling needs and running example

In this section we illustrate MBSA concepts on a simple powertrain model, consisting of two batteries providing power to two electric engines. The failure condition is the loss of both engines. A battery component, shown in Figure 1, has two internal states Ok and Fail, an exponential failure delay distribution of parameter *lamB*. It produces a data-flow `power` representing the power failure mode, Ok in the Ok state and Fail in the Fail state. An engine component has two states Ok and Fail, an exponential failure delay distribution of parameter *lamE*. It produces a data-flow `thrust` representing the thrust failure mode, equal to its input `power` in the Ok state, and to Fail in the Fail state.

Components encapsulate states and guarded transitions behind a data-flow interface. Data-flow connections shown in Figure 2 model how failure modes propagate from batteries, to engines, to the failure condition observer through the system. Each engine's power input is connected to both batteries using an OR operator (produces Ok if one of the inputs is Ok, Fail otherwise). The engines' `thrust` outputs are connected to a failure condition observer

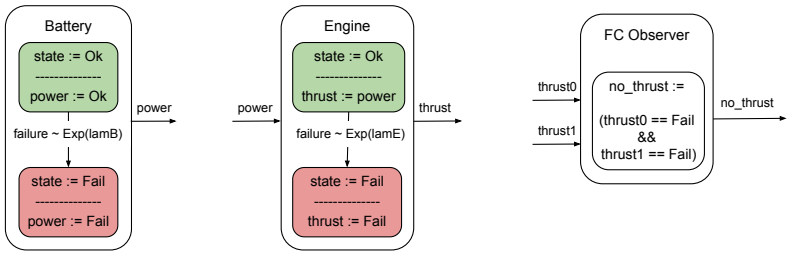


Figure 1 Powertrain example: battery, engine, failure condition observer components.

monitoring the loss of thrust on both engines. In the initial state shown in Figure 2(a), all components are in the Ok state and all power and thrust data-flows are Ok. The state in Figure 2(b) is reached after the failure of the first battery. Since the second battery is still Ok, engines still receive power and produce thrust and the failure condition is not triggered. The state in Figure 2(c) is reached after the failure of the second battery, which causes a loss of power for both engines and loss of thrust, despite the engines being in the Ok state. The failure condition is triggered as a result.

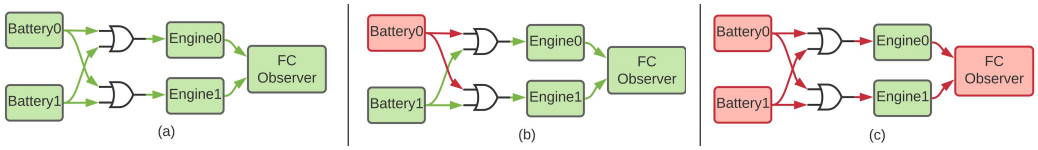


Figure 2 Powertrain example: state and flow updates after Battery0 and Battery1 failure events.

With Alpacas our goal is to formalize such a model in a generic way, where the number of engines and batteries are parameters, and where the topology of the power delivery connections between them is also a parameter of the model. This form of genericity affects the model's hierarchy as well as the topology of the data-flow network. We also want the concrete representation of failure states and failure modes of the engines and batteries to be parameters, as well as the delay distribution parameters of the corresponding events. By combining concepts from stochastic guarded transition systems and generic types, typeclass polymorphism and higher-order concepts from functional programming we can achieve this genericity. This genericity is the basis needed for genuine design-space formalization and exploration.

4 The Alpacas domain-specific language

This section presents the Alpacas DSL, the modeling workflow and the embedding techniques allowing the Scala syntax to be adapted to safety modeling needs. Section 4.1 to Section 4.5 introduce Alpacas constructs using the running example. Section 4.6 details the expressions language of Alpacas. Section 4.7 shows how we extended the Scala syntax for Alpacas.

Code examples with a green background show Alpacas code written by the end-user, and code examples with a red background show internal Alpacas implementation code. These examples are simplified compared to the actual library code, omitting the source mapping code which allows to track filenames, line numbers and Scala variable identifiers, handled using the `sourcecode` library. This implementation of Alpacas is written in Scala 3.0. Listing 1 presents the Alpacas encoding of the powertrain running example of section 3, which is later detailed in sections 4.1 to 4.5.

```
enum Failure derives Lifted {
  case Ok
  case Fail
}

import Failure.*

given Ord[Failure] with {
  def lt(x: Failure, y: Failure): Boolean = x == Ok && y == Fail
}

class Battery extends Component {
  val state   = State[Failure](init = Ok)
  val power   = OutFlow[Failure]
  val failure = Event(Exponential(1E-5))
  val repair  = Event(Dirac(5), weight = 1.0)
  assertions { power := state }
  transitions {
    When(failure) If state === Ok   Then {state := Fail}
    When(repair)  If state === Fail Then {state := Ok}
  }
}

class Engine extends Component {
  val state   = State[Failure](init = Ok)
  val thrust  = OutFlow[Failure]
  val power   = InFlow[Failure]
  val failure = Event(Exponential(1E-5), policy = Policy.Memory)
  val repair  = Event(Dirac(1))
  assertions {
    thrust := If (power === Ok && state === Ok) Then Ok Else Fail
  }
  transitions {
    When(failure) If(state === Ok && power === Ok) Then {state := Fail}
    When(repair)  If(state === Fail)               Then {state := Ok}
  }
}

type Batteries = Vector[Battery]; type Engines = Vector[Engine]
type Wiring    = (Batteries, Engines) => Assertions

class Powertrain(wiring: Wiring, n: Int) extends Component {
  val batteries = Subs(n)(Battery())
  val engines   = Subs(n)(Engine())
  val observer  = OutFlow[Boolean]
  val ccf       = Event(Exponential(1E-7))
  assertions {
    wiring(batteries, engines)
    observer := engines.map(_.thrust === Ok).reduce(_&&_)
  }
  transitions {
    Sync(ccf) With { batteries.map(_.failure.hard).reduce(_&_) }
  }
}

def one2one(b: Batteries, e: Engines): Assertions =
  e.map(_.power) := b.map(_.power)

def one2all(b: Batteries, e: Engines): Assertions =
  for (eng <- e) eng.power := b.map(_.power).reduce(_ min _)

val powertain121    = Powertrain(one2one, 2)
val powertrain12all = Powertrain(one2all, 2)
```

Listing 1 ALPACAS modeling of the powertrain example (cf Figure 1 for graphical view).

4.1 Lifting types, declaring components, state and flow variables

Alpacas supports Scala's built-in *Boolean*, *Int* and *Double* types. Any Scala enumerated type can be lifted in the DSL and used to model component states and failure modes. Lines 1-4 of Listing 1 define a `Failure` enum with two values `Ok` and `Fail`, and lift it in Alpacas space using the `derives Lifted` clause. The mechanism allowing this syntax will be detailed in Section 4.7.

It is possible to define ordering relations on user-defined types in order to use the DSL's relational operators $<$, $\leq$, $\geq$, $>$, min, max in guards and data-flow expressions. Orderings facilitate the definition of generic failure conditions or failure mode consolidation logic that only require to know if a failure mode is worse or better than another, without knowing exactly the individual failure modes. Lines 8-10 of Listing 1 define failure mode `Ok` to be strictly lesser than failure mode `Fail`.

Alpacas allows to specify SGTS in a modular and composable way, and to derive a flat SGTS automatically. All user-defined safety components are represented as Scala classes extending an abstract `Component` class provided by the Alpacas library. Components encapsulate state and flow variable declarations, event declarations, groups of transitions and flow assertions and have a strongly typed defined data-flow interface.

The model structure is captured using object-orientation (classes) and composition. Components can be instantiated inside other components using their constructors and the `Sub` statement. Vectors of sub-components are declared with the `Subs` statement where the size of the vector is provided as first argument (See lines 43-44 in Listing 1). The hierarchy of an Alpacas model represents the system's static architecture.

Components contain either state variables declared by specifying their type and initial value with `State[Type](initial)`, or oriented flow variables declared by specifying their type and interface orientation with `OutFlow[Type]` or `InFlow[Type]`. In lines 25-27 of Listing 1, we define the variables for the `Engine` component: the state variable of type `Failure` and initial value `Ok` represents the intrinsic failure state of the component, the `power` input flow of type `Failure` represents the status of the power supply, and the `thrust` output flow represents the status of the thrust provided by the engine. Listing 2 shows how to declare vectors of variables with the keywords `States`, `InFlows` and `OutFlows`, which take the vector size as parameter.

```
1 class VectorExample extends Component {
2   val state   = States[Failure](init = Ok)(4)
3   val inputs  = InFlows[Failure](4)
4   val outputs = OutFlows[Failure](4)
5 }
```

Listing 2 Vectors of variables.

4.2 Declaring flow assertions

Flow assertions define the flow variables in function of the state variables. Each component must define all its locally declared output flow variables, as well as all input flow variables of its sub-components. Line 31 of Listing 1 defines the `thrust` output of the `Engine` component to be `Ok` if the engine doesn't have an internal failure and receives nominal power supply. Alpacas offers an overloaded flow definition operator := which works with equally sized vectors as left and right hand sides, as shown in line 57 of Listing 1. Functional iterators or for comprehensions can also be used to define vectors of flows point-wise, as shown in line 60 of Listing 1. A flow variable can be defined using any expression over flow or state variables as long as no cyclic flow dependency is introduced. Cyclic definitions are checked by the tool and reported to the user as hard errors (see Section 6.1).

4.3 Declaring transitions and synchronizations

Guarded transitions specify how the system state evolves over time. They are labeled by an event, and composed of a guard (a Boolean expression that must be true for the transition to be fired), and a set of state assertions (specifying how state variables are modified when the transition is fired).

Events represent random faults or deterministic system reactions and carry their delay distribution. Random faults are usually modeled using Exponential distributions, Weibull distributions, etc. Deterministic failure propagation or functional reactions of the system are modeled as events with Dirac distributions. When a transition is fireable, its firing delay is sampled from the distribution associated to its event (Dirac distributions produce a deterministic value). The default behaviour is to sample a new delay every time the transition becomes fireable, but it is also possible to store the delay when the transition stops being fireable and to use the stored delay value the next time it becomes fireable. This is called the *Memory* policy, it is useful to model components that wear out during their use. In line 28 of Listing 1, the failure event for engines is declared with a Memory policy, to model that if the engine is shut down because of a battery failure, when the battery is repaired the engine has the same remaining life as when it stopped being powered.

In order to support common cause modeling, Alpacas offers event synchronization constructs, which express that two or more events can occur simultaneously as a consequence of another event named the *common cause*. The synchronized events can be either:

- *hard-synchronized*: all guards have to be true for the synchronized transition to be fired,
- *soft-synchronized*: at least one of the guards has to be satisfied for the synchronized transition to be fired. The state variables of soft-synchronized transitions are updated only if their guard was satisfied.

Line 52 of Listing 1 shows the hard-synchronization of the failures of two different batteries under a common cause failure event `ccf` (declared on line 46) that models a failure event affecting both engines at the same time (for instance a fire event, a lightning strike event, etc.). The `repair` event of the `Battery` component is declared with `Dirac(5)` delay distribution and `weight` parameter of 1.0 on line 16. The weight parameter is used to handle tie breaks between concurrent events. Here, following a `ccf` event, both batteries' `repair` events will be in concurrency. Tie breaks are achieved by selecting sampling a categorical distribution built from the from the weights of the concurrent events, here such that $p(\texttt{batteries(0).repair}) = p(\texttt{batteries(1).repair}) = \frac{1.0}{1.0+1.0} = 0.5$.

Line 28 of Listing 1 shows how to declare an Exponential distribution for the `failure` event of an engine, and line 29 a Dirac distribution for the functional `repair` event.

4.4 Specifying failure conditions

Any Boolean-valued data-flow of the model can be used as failure condition. For instance, the `observer` flow defined on line 49 of Listing 1 becomes false when the thrust of at least one engine is not `Ok`. Such definitions are usually placed in *observer components*, which are instantiated alongside the other components in the system. Several observers can exist in the system, however analyses take a single failure condition as parameter. Minimal sequences generation searches for event scenarios falsifying the condition. Unreliability analysis estimates the probability of this data-flow becoming false over some mission time T.

4.5 Parameters, type parameters, higher-order parameters

Component constructors can take parameters, allowing for instance to parameterize the number of sub-components or the number of state or flow variables of the component. Functional iterators (map, fold, reduce, ...) and vector assertions allow to define size-agnostic expressions, guards, assertions sets, etc.

Line 42 of Listing 1 declares the Powertrain Component, parameterized by the number of engines and batteries. Batteries and engines are declared as vectors of identical size on lines 43-44. Their data-flow connections are defined by a higher-order `wiring` parameter of type `Wiring`. The `Wiring` type, declared on line 40, is a function type taking `Batteries` and `Engines` vector inputs and producing an implicit function type *Assertions* (provided by the ALPACAS library) as output. The observer expression is defined as the conjunction of all engines providing thrust using the `reduce` iterator. Wiring schemes 1-to-1 and 1-to-all are defined respectively on lines 56-57 and 59-60. Two system variants with two engines and batteries and different wiring schemes are created using the `Powertrain` constructor on lines 62 and 63.

Type-class polymorphism allows to abstract over failure modes and to define generic flow aggregation logic, as shown in the voter example of Listing 3.

```
1 class Voter[A:Lifted:Ord](n: Int) extends Component {
2   val inputs = InFlows[A](n)
3   val output = OutFlow[A]
4   assertions { output := inputs.reduce(_ max _) }
5 }
```

Listing 3 A generic voter component.

The example in Listing 4 shows how to use a trait and self-type annotation to define a reusable unit of behaviour. Using this trait we could for instance factor the failure logic between `Engine` and `Battery` components.

```
1 trait CanFail(lambda: Double) { self: Component =>
2   val state = State[Failure](init = Ok)
3   val fail  = Event(Exponential(lambda))
4   transitions { When (fail) If (state === Ok) Then { state := Fail } }
5 }
6 class Engine extends Component with CanFail(lambda = 1E-7) { /* ... */ }
7 class Battery extends Component with CanFail(lambda = 1E-5) { /* ... */ }
```

Listing 4 Using traits to encapsulate reusable behaviour.

4.6 Abstract syntax for expressions

We use the initial algebra encoding approach for ALPACAS. Expressions are represented by abstract syntax trees defined inductively by a number of variants. Variants include flow variables, state variables, literal constants and constructors for all supported operations. The full abstract syntax is given below:

```
Expr ::= Const(value) | Svar(ident) | Fvar(ident) | Eq(Expr, Expr) |
         Ite(Expr, Expr, Expr) | Lt(Expr, Expr) | Un(Unop, Expr) |
         NumBin(NumBinop, Expr, Expr) | LogBin(LogBinop, Expr, Expr);

LogBinop ::= And | Or;  NumBinop ::= Add | Sub | Mult | Div;  Unop ::= Neg;
```

The following rules define well-typed expressions, where T is a generic type variable:

$$\frac{v \text{ of type } T}{Const(v) : T} \qquad \frac{s \text{ state variable of type } T}{Svar(s) : T} \qquad \frac{f \text{ flow variable of type } T}{Fvar(f) : T}$$

$$\frac{e_1 : T \quad e_2 : T}{Eq(e_1, e_2) : Boolean} \qquad \frac{c : Boolean \quad e_1 : T \quad e_2 : T}{Ite(c, e_1, e_2) : T}$$

The other constructs of the abstract syntax are defined only for some type-classes. We now present these type-classes and the corresponding typing rules.

Numeric is the type-class for numeric operations (addition, subtraction, multiplication and division), with typing rule:

$$\frac{e_1 : T \quad e_2 : T \quad NumBinop \in Add|Sub|Mult|Div \quad Numeric[T]}{NumBin(NumBinop, e_1, e_2) : T}$$

Logic is the type-class for Boolean operations (conjunction, disjunction and negation), with typing rules:

$$\frac{e_1 : T \quad e_2 : T \quad LogBinop \in And|Or \quad Logic[T]}{LogBin(LogBinop, e_1, e_2) : T} \qquad \frac{e : T \quad Logic[T]}{Un(Neg, e) : T}$$

Ord is the type-class of ordered types, with typing rule:

$$\frac{e_1 : T \quad e_2 : T \quad Ord[T]}{Lt(e_1, e_2) : Boolean}$$

The expression language and typing constraints are implemented in Scala 3 using the generalized algebraic datatype (GADT) shown in Listing 5. An implicit conversion for lifting Scala constants to expressions is also provided. Alpacas expressions requiring a given type-class can only be constructed if an implicit type-class instance can be derived by the compiler for this type. This ensures that only well-typed expressions can be represented in the DSL. The type-checking of Alpacas expressions is performed by the Scala compiler and type errors are highlighted in the IDE used for editing the models.

```
enum Expr[T] {
  case Const(value: T) extends Expr[T]

  case Svar(uid: StateId, init: T) extends Expr[T]

  case Fvar(uid: FlowId) extends Expr[T]

  case Eq(l: Expr[T], r: Expr[T]) extends Expr[Boolean]

  case Ite(c: Expr[Boolean], t: Expr[T], e: Expr[T]) extends Expr[T]

  case Lt(l: Expr[T], r: Expr[T])(using Ord[T]) extends Expr[Boolean]

  case NumBin(b: NumBinop, l: Expr[T], r: Expr[T])(using Numeric[T])
    extends Expr[T]

  case LogBin(b: LogBinop, l: Expr[T], r: Expr[T])(using Logic[T])
    extends Expr[T]

  case Un(u: LogUnop, e: Expr[T])(using Logic[T]) extends Expr[T]
}

given [T]: Conversion[T, Expr[T]] with {
  def apply(t:T): Expr[T] = Expr.Const(t)
}
```

Listing 5 Scala GADT for Alpacas expressions.

4.7 Syntax extensions

As seen in the code examples of sections 4.1 to 4.3, Alpacas provides constructs allowing to declare variables, assertions, transitions and expressions with a natural syntax. We use Scala 3's context abstraction capabilities to perform the required book-keeping of state and flow variables, events, assertions and transition declarations without adding clutter for the end-user. The code of Listing 6 presents the `State` variable constructor (`InFlow` and `OutFlow` variable constructors definitions are similar). This constructor takes an implicit argument of type `StateVarSet` from the surrounding `Component` instance, and creates a new `Expr.Svar` instance representing a state variable, adds it to the set of variables of the component, and returns it.

```
object State {
  def apply[T](init: T)(using svar: StateVarSet): Expr.Svar[T] =
    val res = new Expr.Svar[T](StateId(), init)
    svar += res
    res
}
```

Listing 6 State variable constructor.

To group assertions declarations in an `assertions` block, we use context functions and Odersky's *builder pattern* [40]. The builder pattern allows to build data structures with a declarative syntax, hiding side effects performed by builder methods. Multiple builder patterns can be nested by introducing intermediary builder methods.

Listing 7 shows the `assertions` builder method. It takes an implicit `ComponentBuilder` argument, used to perform all book-keeping declarations and definitions found inside a component, that is only available when in a surrounding `Component` instance. The field `flowAssertionBuilder` of the `builder` object is placed in the implicit scope of the `assertions` method to make it available to the `:=` assertion definition operator (itself defined as an extension method in the derived `Lifted` instance, see Listing 9). The `init` argument of the `assertions` method, with implicit function type `FlowAssertionBuilder ?=> Unit`, is provided by the user as a block containing flow assertions. Nesting the `FlowAssertionBuilder` inside the `ComponentBuilder` ensures that a compile-time error occurs when attempting to define flow assertions outside of an `assertions` builder method.

```
def assertions(init: FlowAssertionBuilder ?=> Unit)
              (using builder: ComponentBuilder) =
  given FlowAssertionBuilder = builder.flowAssertionBuilder
  init
```

Listing 7 assertions function for the builder pattern defining flow assertions.

The `transitions` builder uses three levels of nesting: the `transitions` builder method takes an implicit `ComponentBuilder`, which contains a `TransitionBuilder` object provided to the `When(e) If(g) Then { v := expr }` builder construct, which itself contains a `StateAssertionsBuilder` object provided to the `:=` state assertion definition operator.

`Lifted`, shown in Listing 8, is the type-class for types that can be lifted to Alpacas expressions. It allows to compare expressions using the equality `===` operator. The `:=` overloaded operator allows to define state variables in transitions (cf. Section 4.3) and to define flow variables in assertions (cf. Section 4.2).

```
trait Lifted[T] {
  extension (x: Expr[T])
    def === (y: Expr[T]): Expr[Boolean]

  extension (x: Expr.Svar[T])
    def := (y: Expr[T]) (using a: StateAssertionBuilder): Unit

  extension (x: Expr.Fvar[T])
    def := (y: Expr[T]) (using a: FlowAssertionBuilder): Unit
}
```

Listing 8 Lifted type-class.

Automatic type-class derivation is used to relieve the user from manually defining the type-class instance (as shown in Section 4.1). For equality, the operator `===` lifts the comparison to an expression. The polymorphic variable assignment operators `:=` takes implicit `FlowAssertionBuilder` and `StateAssertionBuilder` and adds the corresponding assertion to it.

```
object Lifted {
  def derived[T]: Lifted[T] = new Lifted[T] {
    extension (x: Expr[T])
      def === (y: Expr[T]): Expr[Boolean] = Expr.Eq(x, y)

    extension (x: Expr.Svar[T])
      def := (y: Expr[T]) (using a: StateAssertionBuilder): Unit =
        a += StateAssertion(x, y)

    extension (x: Expr.Fvar[T])
      def := (y: Expr[T]) (using a: FlowAssertionBuilder): Unit =
        a += FlowAssertion(x, y)
  }
}
```

Listing 9 Derived instance of type-class Lifted.

Type-classes `Numeric`, `Logic` and `Ord` are implemented using generic traits defining the necessary operations on an abstract type. We have other type-classes defining the corresponding operations on Alpacas Expressions as extension methods, and we use type-parametric givens to automatically derive instances of these type-classes.

Listing 10 shows the `Ord` syntax extensions for expressions. The user provides an instance of type-class `Ord` for lifted type `T` (see Section 4.1). The type-class `DSLord` provides syntax extensions for expressions of the `Ord` type, and the corresponding type-parametric given ensures `DSLord` instances can be derived from `Ord` instances.

```
trait Ord[T:Lifted] {
  def lt(x: T, y: T): Boolean
}

trait DSLord[T: Lifted] {
  extension (x: Expr[T])
    def <   (y: Expr[T]): Expr[Boolean]
    def >   (y: Expr[T]): Expr[Boolean] = !(x < y) && !(x === y)
    def <=  (y: Expr[T]): Expr[Boolean] = x < y || x === y
    def >=  (y: Expr[T]): Expr[Boolean] = !(x < y)
    def min (y: Expr[T]): Expr[T]       = If (x < y) Then x Else y
    def max (y: Expr[T]): Expr[T]       = If (x < y) Then y Else x
}

given [T:Lifted:Ord]: DSLord[T] with {
  extension (x: Expr[T])
    def < (y: Expr[T]): Expr[Boolean] = Expr.Lt(x, y)
}
```

Listing 10 Type-class mechanism for ordered types.

For conditional flow selection, we use functions and infix methods to produce `IfThenElse` expressions as presented in Listing 11. Due to Scala parsing rules, the parenthesis are mandatory around the conditional but optional around the branches

```
def If(c: Expr[Boolean]): Ift = Ift(c)

case class Ift(c: Expr[Boolean]){
  def Then[T] (t: Expr[T]): IfThent[T] = IfThent(c, t)
}

case class IfThent[T](c: Expr[Boolean], t: Expr[T]){
  def Else (e: Expr[T]): Expr[T] = Expr.Ite(c, t, e)
}
```

Listing 11 Implementation of conditional statements.

5 Stochastic guarded transition systems

The semantics of an ALPACAS model is given by a Stochastic Guarded Transition System (SGTS). Our version of SGTS is largely inspired from [42, 9]. This formalism allows to model dynamic, repairable and re-configurable systems. From [42, 9], we reuse the notions of state and flow variables, Restart and Memory transitions, event concurrency resolution mechanisms and event synchronization mechanisms. However, we only accept causal systems and we add the notion of Urgent events. Urgent events have priority over all other events.

5.1 Definitions

▶ **Definition 1** (Stochastic Guarded Transition System). *A* Stochastic Guarded Transition System *is a tuple:*

$$SGTS = \langle S, F, A_F, T, E \rangle \tag{1}$$

Where:

- S *is a vector of typed* ***state variables****. Each state variable has an initial value* v_{init}*;*
- F *is a vector of typed* ***flow variables*** *propagating failure modes through the system;*
- A_F *is a set of* ***flow assertions*** *of the form* $v := expr$*, with* $v \in F$ *and* $expr$ *an expression over state and flow variables defining* v *at all times;*
- T *is a set of* ***guarded transitions*** *of the form* $g \xrightarrow{e} A_S$ *where:*
  - e *is an* ***event****, the* ***trigger*** *of the transition;*
  - g *is a Boolean expression over state and flow variables, the* ***guard*** *of the transition;*
  - A_S *is a set of* ***state assertions*** *of the form* $v := expr$ *with* $v \in S$ *and* $expr$ *an expression over sate and flow variables, describing updates applied to state variables when the transition is fired.*

Transitions are of three different types, which condition the way they are scheduled in the system's behaviour:

- ***Urgent*** *transitions have priority over all other transitions and are fired immediately after their guard becomes true, without delay.*
- ***Restart*** *transitions have an associated firing delay distribution* $dist(e)$ *and an optional real-valued weight parameter* $W(e)$*. The firing delay is sampled from the distribution each time a state where the guard is true is reached.*

- ***Memory*** *transitions have an associated firing delay distribution* $dist(e)$ *and an optional real-valued weight parameter* $W(e)$*. The firing delay is sampled the first time the guard becomes true, and sampled again only after the transition is fired, when the guard becomes true again. When the guard becomes false, the current delay value is saved and restored the next time the guard becomes true.*

The different transition types entail a partition of the set of transitions $T = T_U \cup T_R \cup T_M$*;* $E = E_U \cup E_R \cup E_M$ *is the set of events, partitioned by event type.*

Example 2 shows the flat SGTS encoding of the powertrain running example presented in Listing 1. The If-Then-Else expressions appearing in flow definitions are the result of rewriting the `min` operator in terms of core operators. The common cause `ccf` transition was rewritten using the rules presented in Section 5.3.

▶ **Example 2** (Powertrain SGTS, one2all wiring)**.**

$$
\begin{aligned}
S &= \{\ b_0.state(init := Ok),\ b_1.state(init := Ok),\ e_0.state(init := Ok),\ e_1.state(init := Ok)\ \}\\
F &= \{\ observer,\ b_0.power,\ b_1.power,\ e_0.power,\ e_0.thrust,\ e_1.power,\ e_1.thrust\ \}\\
A_f &= \{\ b_0.power := b_0.state,\ b_1.power := b_1.state,\\
&\quad e_0.power := Ite(b_0.power < b_1.power,\ b_0.power,\ b_1.power),\\
&\quad e_1.power := Ite(b_0.power < b_1.power,\ b_0.power,\ b_1.power),\\
&\quad e_0.thrust := Ite(e_0.power = Ok \wedge e_0.state = Ok,\ Ok,\ Fail),\\
&\quad e_1.thrust := Ite(e_1.power = Ok \wedge e_1.state = Ok,\ Ok,\ Fail),\\
&\quad observer := e_0.thrust = Ok \wedge e_1.thrust = Ok\ \}\\
T_R &= \{\ b_0.state = Ok \wedge b_1.state = Ok \xrightarrow{ccf \sim Exp(1e-7)} \{b_0.state := Fail,\ b_1.state := Fail\},\\
&\quad b_0.state = Ok \xrightarrow{b_0.failure \sim Exp(1e-5)} \{b_0.state := Fail\},\\
&\quad b_0.state = Fail \xrightarrow{b_0.repair \sim Dirac(5), W=1} \{b_0.state := Ok\},\\
&\quad b_1.state = Ok \xrightarrow{b_1.failure \sim Exp(1e-5)} \{b_1.state := Fail\},\\
&\quad b_1.state = Fail \xrightarrow{b_1.repair \sim Dirac(5), W=1} \{b_1.state := Ok\},\\
&\quad e_0.state = Fail \xrightarrow{e_0.repair \sim Dirac(1)} \{e_0.state := Ok\},\\
&\quad e_1.state = Fail \xrightarrow{e_1.repair \sim Dirac(1)} \{e_1.state := Ok\}\ \}\\
T_M &= \{\ e_0.state = Ok \wedge e_0.power = Ok \xrightarrow{e_0.failure \sim Exp(1e-5)} \{e_0.state := Fail\},\\
&\quad e_1.state = Ok \wedge e_1.power = Ok \xrightarrow{e_1.failure \sim Exp(1e-5)} \{e_1.state := Fail\}\ \}\\
T_U &= \{\ \}
\end{aligned}
$$

The expression language used in assertions (already detailed in section 4.6) supports Boolean expressions, integer and floating point numeric expressions as well as equality checks over user-defined enumerations types. We only consider well typed expressions and assertions. A *total valuation* α is a total function over $S \cup F$ assigning a value to each state variable and flow variable, that can be decomposed into a state variable valuation α_S and a flow variable valuation α_F. We assume a function *eval* which evaluates an expression in the context of a valuation α. In a given state, the valuation α_S is defined relative to the previous state's total valuation α, whereas the valuation α_F is defined relative to the current α_S.

We assume that A_F contains a definition for each flow variable. A flow variable v depends on a state or flow variable v' if v' occurs in the expression defining v in A_F. We only consider *causal* systems where flow dependency is acyclic, so that there exists a topological ordering of flow variables allowing to evaluate all flow assertions in a single pass to obtain a flow valuation $\alpha_F = propagate(\alpha_S)$. A transition $g \xrightarrow{e} A_S$ is *fireable* in the context of a total valuation α if and only if $eval(g, \alpha)$ is true. We say that a valuation α is *stable* if no urgent transition is fireable in α, and *unstable* otherwise. Urgent transitions allow to model immediate feedback loops while preserving causality: a cycle in data-flow definitions is broken by introducing a

stateful element in the cycle and delaying flow propagation to the next logical step using urgent transitions. Restart transitions allow to model random failure events for memoryless components for which state history has no influence. Memory transitions allow to model random failures of components for which the state history has an influence.

5.2 Stochastic timed trace semantics

▶ **Definition 3** (Timed Trace). *The semantics of stochastic guarded transition system is given by timed traces of the form:*

$$TimedTrace = S_0 \xrightarrow{e_0} S_1 \cdots \xrightarrow{e_{i-1}} S_i \xrightarrow{e_i} S_{i+1} \ldots \xrightarrow{e_{n-1}} S_n \tag{2}$$

A trace is a sequence of states S_i connected by Restart or Memory transitions where

$S = \langle \overline{\alpha}, \alpha, \Sigma, Mem, t \rangle$

is such that:

- $\overline{\alpha}$ *is a (possibly unstable) valuation,*
- α *is a stable valuation,*
- $\Sigma : E_R \cup E_M \to \mathbb{R}^+ \cup \{+\infty\}$ *is an* event schedule *associating a firing delay to each restart and memory event,*
- $Mem : E_M \to \mathbb{R}^+$ *is an* event delay memory *associating a memorized delay to each memory event,*
- t *is a positive real value representing the timestamp of the state.*

Firing a transition $g \xrightarrow{e} A_S$ in the context of a stable or unstable valuation α (decomposed in α_S and α_F) yields a new valuation α' decomposed in α'_S and α'_F defined by:

$$\alpha'_S(v) = \begin{cases} eval(expr, \alpha) & \text{if } \{v := expr\} \in A_S \\ \alpha_S(v) & \text{otherwise} \end{cases} \tag{3}$$

$$\alpha'_F = propagate(\alpha'_S) \tag{4}$$

When in a state S_i, the Restart or Memory transition to fire is the one with the smallest delay in the event schedule, $e_i = \text{argmin}(\Sigma_i)$. If several events have the same minimum delay value, the weight values of the concurrent events are used to break the tie. A categorical distribution is created such that $p(e) = \frac{W(e)}{\sum_{e \in \text{argmin}(\Sigma_i)} W(e)}$, and the event e_i is sampled from this distribution.

The (possibly unstable) valuation $\overline{\alpha}_{i+1}$ is the result of firing the transition associated to event e_i in the stable valuation α_i.

The stable valuation α_{i+1} is determined by exploring all possible interleavings of fireable urgent transitions starting from $\overline{\alpha}_{i+1}$, transitively across unstable valuations. If all interleavings lead to the same stable valuation α_{i+1}, it is taken as the stable valuation for the successor state S_{i+1}, otherwise the trace is considered invalid.

- For each Restart event e, the schedule at state $i+1$ is defined depending on whether e is the event e_i that was fired in state i or not, and on its fireability in states i and $i+1$:

$e = e_i$	$\mathit{fireable}(e, \alpha_i)$	$\mathit{fireable}(e, \alpha_{i+1})$	$\Sigma_{i+1}(e)$
$\top$	$\top$	$\top$	$d \sim \mathit{dist}(e)$
$\top$	$\top$	$\bot$	$+\infty$
$\bot$	$\top$	$\top$	$\Sigma_i(e) - \Sigma_i(e_i)$
$\bot$	$\top$	$\bot$	$+\infty$
$\bot$	$\bot$	$\top$	$d \sim \mathit{dist}(e)$
$\bot$	$\bot$	$\bot$	$+\infty$

- For each Memory event e, the schedule and memory functions at state $i+1$ are defined depending on whether e is the event e_i that was fired in state i or not, on its fireability in states i and $i+1$, and on the value of its delay memory in state i:

$e = e_i$	$\mathit{fireable}(e, \alpha_i)$	$\mathit{fireable}(e, \alpha_{i+1})$	$\mathit{Mem}_{i+1}(e)$	$\Sigma_{i+1}(e)$
$\top$	$\top$	$\top$	$d \sim \mathit{dist}(e)$	$\mathit{Mem}_{i+1}(e)$
$\top$	$\top$	$\bot$	$d \sim \mathit{dist}(e)$	$+\infty$
$\bot$	$\top$	$\top$	$\Sigma_i(e) - \Sigma_i(e_i)$	$\mathit{Mem}_{i+1}(e)$
$\bot$	$\top$	$\bot$	$\Sigma_i(e) - \Sigma_i(e_i)$	$+\infty$
$\bot$	$\bot$	$\top$	$\mathit{Mem}_i(e)$	$\mathit{Mem}_{i+1}(e)$
$\bot$	$\bot$	$\bot$	$\mathit{Mem}_i(e)$	$+\infty$

- $t_{i+1} = t_i + \Sigma_i(e_i)$ (the time progresses by the fired event's delay value).

The initial state S_0 of a timed trace is defined by:

- $\overline{\alpha}_{S0}(v) = v_{init}$ for all state variables,
- $\overline{\alpha}_{F0}(v) = \mathit{propagate}(\overline{\alpha}_{S0}(v))$,
- α_0 is obtained by exploring all interleavings of Urgent events starting from $\overline{\alpha}_0$ as described above,
- For each Restart event e:

$$\Sigma_0(e) = \begin{cases} d \sim \mathit{dist}(e) & \text{if } \mathit{fireable}(e, \alpha_0) \\ +\infty & \text{otherwise} \end{cases}$$

- For each Memory event e:
  - $\mathit{Mem}_0(e) = d \sim \mathit{dist}(e)$,
  - $\Sigma_0(e) = \begin{cases} \mathit{Mem}_0(e) & \text{if } \mathit{fireable}(e, \alpha_0) \\ +\infty & \text{otherwise} \end{cases}$
- $t_0 = 0$

5.3 Event synchronizations

It is possible to define synchronizations of several Restart and Memory events (but not Urgent events) with another event called the *common cause event*. The common cause event can have its own delay distribution and weight parameter.

▶ **Definition 4** (Synchronization). *A synchronization has the form:*

$$(e \ : \ a_1.\mathit{hard} \ \& \ \cdots \ \& \ a_m.\mathit{hard} \ \& \ b_1.\mathit{soft} \ \& \ \cdots \ \& \ b_n.\mathit{soft}) \xrightarrow{g} A_S$$

Where

- *e is the common cause event,*
- $\{a_i.hard \mid 0 \leq i \leq m\}$ *are the* mandatory events *of the synchronization,*
- $\{b_i.soft \mid 0 \leq i \leq n\}$ *are the* optional events *of the synchronization,*
- *g is a (possibly true) guard,*
- A_S *is a (possibly empty) set of state assertions.*

The semantics of a synchronization is defined by translation to the core formalism. We assume that the transitions corresponding to synchronized events are already rewritten to standard transitions if they were synchronized transitions, so that we have a set of mandatory transitions of the form: $M = \{h_1 \to A_{s_1}, ..., h_l \to A_{s_l}\}$ and a set of optional transitions of the form: $O = \{j_1 \to B_{s_1}, ..., j_n \to B_{s_n}\}$.

We denote by *If g Then* B_s the set of state assertions B_s where each assertion $v := expr$ is rewritten to $v :=$ *If g Then e Else v*. The translation is defined as follows:

- **Case $l > 0$**: The synchronization rewrites to:

$$h_1 \,\&\&\, ... \,\&\&\, h_l \xrightarrow{e} A_{s_1} \cup ... \cup A_{s_l} \cup \mathit{If}\ j_1\ \mathit{Then}\ B_{s_1} \cup .. \cup \mathit{If}\ j_n\ \mathit{Then}\ B_{s_n}$$

- **Case $l = 0$ and $n > 1$**: The synchronization rewrites to:

$$j_1 \,||\, ... \,||\, j_l \xrightarrow{e} \mathit{If}\ j_1\ \mathit{Then}\ B_{s_1} \cup .. \cup \mathit{If}\ j_n\ \mathit{Then}\ B_{s_n}$$

- **Case $l = 0$ and $n = 1$**: The synchronization rewrites to:

$$\mathit{true} \xrightarrow{e} \mathit{If}\ j_1\ \mathit{Then}\ B_{s_1}$$

5.4 Instability, Zeno phenomena and other issues

The definitions given in the previous sections do not prohibit ill-conditioned systems where the following issues occur:

- multiple distinct stable valuations are reachable from a given unstable valuations,
- the system exhibits Zeno behaviour, i.e. can take an infinite number of transitions through unstable valuations, or through stable states or a combination of both in a finite amount of time,
- event concurrency situations which cannot be solved because of a missing weight parameter (which we handle as a modeling error from the user),
- systems with unwanted deadlock states due to synchronizations of transitions with incompatible guards, etc.
- runtime errors in expression evaluation such as arithmetic underflow/overflow, division by zero, etc.

Static analysis or model-checking algorithms allow to detect such issues ahead of time, however in this first version of Alpacas we detect such problems at run-time when exploring event sequences or simulating the system, leaving the more advanced method for future work. Detection is performed by monitoring diverging interleavings of urgent transitions; monitoring for cycles of unstable states; exiting in error if a threshold was exceeded on the number of fired events (including urgent events) without having time progress; exiting in error in case an event without weight parameter is involved in a concurrent race. We also offer an interactive step simulator that allows the user to test the model against their own expectations.

6 Alpacas algorithms

This section presents the main algorithms available in Alpacas allowing to process a model and compute its safety indicators: flattening, basic evaluation and step simulation, minimal cut sequence enumeration, stochastic simulation.

6.1 Translating a hierarchical model to a flat stochastic guarded transition system

Hierarchical models need to be translated to the underlying SGTS representation to be analyzed. Since the hierarchy is flattened in the process, this translation is called `flattening`.

The first part of the flattening is to traverse the structure recursively to collect all variables, assertions and transitions of the model. We store them in adequate structures referencing them by their unique identifiers. We also generate human-readable names for variables and events reflecting to their full path in the component hierarchy.

Then several checks are performed. We use the `cats` library's `Validated` type to accumulate errors of several parallel validation tasks. The first check is for flow definitions: we verify that each component actually defines exactly once all the flows it must define (its output flows and its sub-components' input flows). If it is not the case, we accumulate all errors corresponding to missing or redundant definitions (with variables names and line of declaration) and send back the errors to the user. The second check is for model causality: we verify that the flow dependency is not cyclic. To do this, we generate the graph representing the dependency relation between flow variables defined by flow definitions (we use the `scalagraph` library). The absence of cyclic definitions is verified if and only if every strongly connected component of the graph contains only one node and flow assertions do not create direct self-dependencies. We check this using `scalagraph`, and in case of failure produce an error describing all variables involved in every cyclic component of the dependency graph. If no error is found, we compute a topological ordering on the graph that allows to compute flow variable assignments in sequential order.

Finally, we rewrite synchronizations to standard transitions according to the definitions presented in Section 5.3. This is done thanks to a recursive function that we call on every transition. Every time a synchronization is found, we recursively flatten the synchronized events (that can themselves correspond to synchronizations).

6.2 Transition firing and state updates

The basis of all analyses that can be made on an SGTS is the representation of α_S and α_F valuations and how they are updated to reflect the firing of a transition, moving one step forward in the trace of a valid run of the SGTS.

As described in Section 5.2, firing a transition consists in computing the new state valuation according to the previous total valuation and to the state assertions of the fired transition, followed by computing the flow valuation according to the new state valuation and to all flow assertions in topological order, iterating this process as long as urgent transitions are possible, to finally reach a stable valuation or exit in error if divergent urgent behaviour is detected or Zeno behaviour is detected.

Another important basic function used in all algorithms is the computation of the list of fireable transitions. This is straightforward from the evaluation of all transitions guards in a given state.

These two building blocks allow us to provide an interactive step simulator. When in this mode, the values of variables and fireable transitions in the current assignment are displayed to the user who can manually choose the next transition to fire (instead of using the minimum delay rule of the timed trace semantics). The next state is then displayed (with an option for displaying only the state and flow variable delta with respect to the previous state), so on and so forth until the user stops the simulation. Thanks to the functional immutable data structures backing this simulation mode, the user can undo previous decisions at any point and backtrack in the simulation in order to explore another branch.

6.3 Qualitative indicators

The enumeration of minimal sequences requires to produce traces that lead to a state satisfying a failure condition. To avoid redundancies, only minimal failure scenarios according to a given partial ordering over sequences are considered in safety analysis. We support the most common ordering used in the safety literature, which is the subsequence relation. To generate all possible minimal sequences, we explore the set of possible failure sequences using a bounded breadth-first search algorithm, allowing to generate sequences that are minimal by construction: sequences of size n are naturally explored only after all sequences of smaller sizes are explored. We also avoid visiting extensions of sequences that are already known to satisfy the failure condition.

```
val queue = Queue((immutableInitialState(model), List[EventId]()))
var res: List[List[EventId]] = Nil
while (!queue.isEmpty)
  val (state, seq) = queue.dequeue()
  if (eval(failureCondition, state) && !res.exists(subSequence(_, seq)))
    res = seq::res
  else if (seq.size < maxSize)
      val ftrans = fireable(model, state)
      ftrans.foreach{t =>
        val newSeq = t.id::seq
        if (!res.exists(subSequence(_, seq)))
          val newstate = fire(state, t.id)
          queue.enqueue((newState, newSeq))
      }
res
```

Listing 12 Breadth-first search with online minimization for minimal sequences enumeration.

From the minimal cut sequences we can deduce the minimal cutsets by forgetting the order and eliminating redundancies. If the system is static, this operation doesn't remove any information (the minimal sequences correspond to all permutations of the minimal cutsets), but if it is dynamic, we possibly lose information about the dysfunctional behaviour of the system (the exact ordering of events required to trigger a failure condition), which however translates to safe pessimism for the analysis. Due to the combinatorial explosion of the exploration for large systems, very high order cutsets are often neglected in order to scale the computations on large models. Low order cutsets (up to order 3) are the direct target of regulations and hence have the strongest impact on design decisions, and are the largest contributors to unreliability. Nevertheless, the probability of unexplored scenarios can be soundly approximated by considering they all trigger the failure condition.

We give in Table 1 the output given by the tool for minimal cutsets of the example given in Listing 1. The failure condition is the loss of thrust for one or more engine, the results are as expected: the intrinsic failure of either one engine or the other trigger the failure condition, as does the loss of both batteries, either by the combination of their failure events,

Table 1 MCS for `powertrain12all`.

Order	Minimal cutset
1	ccf
1	engines(0).failure
1	engines(1).failure
2	batteries(0).failure, batteries(1).failure

or by a common cause failure triggering the simultaneous loss of both batteries (a single battery loss is tolerated thanks to the one-to-all wiring). More efficient SAT or SMT-based model-checking techniques can also be used for minimal cutset [21] or minimal sequence enumeration [14], with an explicit time model [1] or without. Our initial focus being on language expressivity, we leave this as future work.

6.4 Quantitative indicators

▶ **Definition 5** (Reliability, Unreliability). *Let t_{fail} be the random variable describing the instant at which system failure occurs.* ***Reliability*** *for a mission time T is defined as the probability that the system failure does not occur in the interval $[0, T]$, knowing that the system is in perfect nominal condition at time 0.* ***Unreliability*** *is the complement of reliability.*

$$R(T) = p(t_{fail} > T), \quad U(T) = 1 - R(T)$$

The reliability of the system can be computed from minimal cutsets using a BDD-based algorithm [41]. We provide an implementation of this algorithm using the JavaBDD library. It relies on the user-specified delay distributions for events (this analysis is offered only if all distributions are specified), and is evaluated for a given mission time T. The computation yields an exact result if it is based on all cutsets for a static system, and becomes a safe under-approximation if the system is dynamic. The computation yields a possibly unsafe approximation for both static and dynamic systems if cutsets of high order are neglected. This BDD-based analysis cannot take dynamic repair or reconfiguration events into account.

Monte-Carlo simulation on the other hand allows to take into account the dynamic repair and reconfiguration of a system without approximation. The Alpacas stochastic simulator allows to sample finite traces of an SGTS and to compute safety indicators on the fly, by directly folding traces using a statistics aggregation function, without storing the traces. We provide aggregators for usual safety indicators such as (un)reliability, availability, mean time between failures, etc. The Monte-Carlo estimates converge in $\frac{1}{\sqrt{\#samples}}$ and high-confidence intervals can be computed based on the empirical sample mean and variance. The Alpacas simulator supports multi-core parallelism thanks to Scala's parallel collections library.

Table 2 gives a comparison of the runtimes and results of the Minimal Cutsets + BDD method vs the Monte-Carlo method for unreliability estimation. Results were obtained on a quad core MacBook Pro 13" 2019 with 16gigs of Ram. For mission times up to 10^3 time units, Minimal Cutsets + BDD and Monte-Carlo results are equal up to the third decimal. The difference on the remaining decimals can be attributed to the natural imprecision of Monte-Carlo methods. For longer mission times, the Monte-Carlo unreliability is lower than the MCS unreliability. This is due to the repairability of the system which is neglected by the Minimal Cutsets + BDD technique. The computation cost for an estimation of the reliability is significantly higher for the Monte-Carlo method, and it increases with the duration of mission time, which is not the case for the Minimal Cutsets + BDD method. However, the cost of preliminary computations needed for each analyzed architecture must be taken

into account. Flattening is necessary for both analyses while the computation of Minimal Custsets and the structure function's BDD are necessary only for the Minimal Custsets + BDD algorithm. For large models, the BDD computation typically becomes the bottleneck.

Table 2 Runtimes (ms) per preprocessing phase, MCS+BDD vs Monte-Carlo (10^5 samples, 95% confidence interval) and runtimes (ms) for Unreliability of `powertrain12all`.

Preprocessing Phase	CPU time
Flattening	377
MCS	13
BDD	18

T	$U(T)$ MCS+BDD	CPU time	$U(T)$ Monte-Carlo	CPU time
10^2	0.0020	< 1	0.00213 ± 0.00003	271
10^3	0.0201	< 1	0.0209 ± 0.0003	266
10^4	0.189	< 1	0.181 ± 0.002	307
10^5	0.919	< 1	0.867 ± 0.001	456

Importance sampling or importance splitting algorithms [29, 15] are well known techniques for rare event estimation that can scale better and converge faster than unbiased Monte-Carlo. However, deriving meaningful importance functions (typically real-valued functions) in our discrete setting requires further research. Recent property-directed algorithms for probabilistic model checking [10] mixing symbolic and quantitative analysis for Markov Processes look very promising, but would need to be generalized to be applicable to Alpacas models (Alpacas models can be semi-Markov and even more general due to the Memory transitions).

7 Design-space exploration for an eVTOL thrust reallocation function

The main objective of this case study is to demonstrate that the Alpacas feature set makes it indeed well suited for safety modeling (including dysfunctional and functional behaviour) and design-space exploration for system architectures involving varying numbers of components, and alternative data-flow connections schemes. Another goal is to illustrate the kind of system design tradeoffs that can be analyzed through design-space exploration.

For this purpose, we chose to model a thrust system for a multi-rotor eVTOL able to tolerate any single fault while preserving safe hovering capability. It requires to compensate thrust loss while preserving thrust symmetry. The approach used for thrust compensation is described in [7]. It consists in shutting down the engine opposite to the failing engine to maintain symmetry with respect to all rotational axes, and to reallocate the missing lift on the remaining engines by increasing (trimming) their default thrust value.

The choice of architecture for this thrust function is not obvious, and requires automatic exploration. We must take into account the failure modes of all components involved: Batteries, Engines, Sensors, and CPUs executing the thrust reallocation logic. Thrust loss can be due to a intrinsic engine failure, or to a failure of the batteries powering the engine. It can also be due to a failure of a sensor triggering a spurious trim. The reallocation logic itself can also be lost due to CPU malfunction, or due to a battery failure, etc. From a cost/reliability trade-off perspective, a design using few engines requires high trim levels and high nominal engine thrust, and hence larger and more powerful engines and batteries, which comes at a cost. A design using more engines requires smaller nominal thrusts and trim levels in single failure cases, possibly cheaper engines, and could tolerate double failures. It has other downsides like wiring complexity and increased weight and it still requires high trim values in double failure scenarios, possibly quickly degrading the health of small engines.

We propose a parametric family of architectures allowing to implement the reconfiguration logic. In this study, we propose a parametric ALPACAS model capturing the design-space, and compute safety indicators for a number of configurations to identify design tradeoffs, and select the safest architecture(s).

Figure 3 shows one of the many possible architectures for the system (engine positions in the picture do not reflect their actual position in the aircraft): 6 batteries, 6 engines, one sensor per engine, dual computing units, dual power redundancy for all components, shared power sources for diagonally opposed engines and sensors, segregated power sources for axially opposed engines.

The design-space to explore is parameterized by the number of batteries, engines and sensors $n \in \{6, 8, 10\}$, by the battery failure rate λ_b, by the sensor failure rate $\lambda_s \in \{1\text{E-}5, 1\text{E-}10\}$, by the default sensor readout when it is not working properly (either optimistic or pessimistic, Boolean parameter *opt*). The engine failure rate is a piecewise constant function of the trim value: λ_0 when in [0%,10%], λ_1 when in [10%, 50%], λ_2 when in [50%,100%]. We model two computing units of failure rates $\lambda_c =$ 1E-10. We model dual redundant power source for engines, sensors and one-to-all wiring for computing units. We consider two power source segregation cases (Boolean parameter *seg*): one where a sensor and its engine have the same power source, another where they use different sources. For $n = 6$, the reconfiguration logic doesn't cover double engine failures as this could yield a situation with only 2 engines functioning (2 are failed and 2 are shutdown) resulting in a loss of control and out of range trim values. For $n \in \{8, 10\}$, the logic does trigger a reconfiguration in case of a double engine failure.

The failure rates and mission time chosen for this study are not realistic. Their relative orders of magnitude were simply chosen to illustrate their influence on reliability, and give the reader an idea of the kind of design decisions that can be studied using ALPACAS models and algorithms.

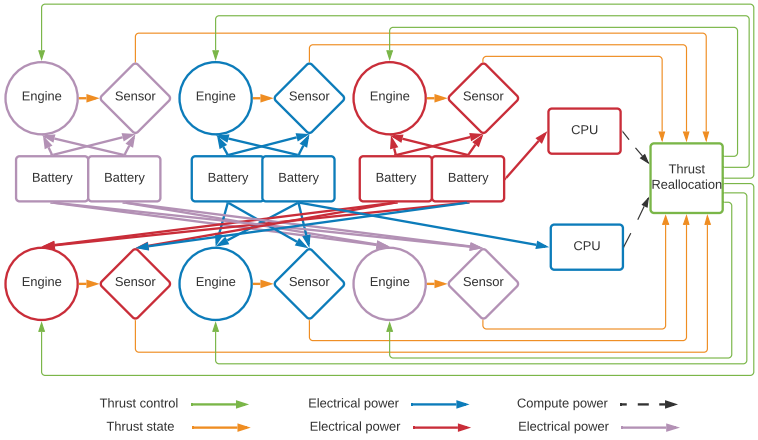


Figure 3 Conceptual diagram of the thrust reallocation system with 6 engines.

The design space exploration results are presented in Table 3. Results are obtained with 100 seconds of computation on a quad core MacBook Pro 13" 2019 with 16gigs of Ram. We use depth-first search for minimal sequences enumeration. We use Monte-Carlo with 100k simulations for unreliability estimation, to properly take the dynamic thrust reallocation behaviour into account. All configurations are immune to single failures (no minimal cutset of

Table 3 Design-space exploration results (mission time 10^3 time units).

n	λ_0	λ_1	λ_2	λ_b	λ_s	*opt*	*seg*	# order 1 mcs	# order 2 mcs	# order 3 mcs	$U(T)$ 95% conf. int.
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	true	false	0	57	6	0.0252 ± 0.0003
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	true	true	0	54	42	0.0253 ± 0.0003
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	false	false	0	120	54	0.0481 ± 0.0006
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	false	true	0	123	6	0.0480 ± 0.0006
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	true	false	0	57	6	0.0248 ± 0.0003
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	true	true	0	54	42	0.0234 ± 0.0003
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	false	false	0	120	54	0.0247 ± 0.0003
6	1.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	false	true	0	123	6	0.0251 ± 0.0003
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	true	false	0	12	320	0.0102 ± 0.0001
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	true	true	0	8	368	0.0106 ± 0.0001
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	false	false	0	0	1568	0.0157 ± 0.0002
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	false	true	0	4	1496	0.0153 ± 0.0002
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	true	false	0	12	320	0.0094 ± 0.0001
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	true	true	0	8	368	0.0094 ± 0.0001
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	false	false	0	0	1568	0.0093 ± 0.0001
8	2.0E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	false	true	0	4	1496	0.0102 ± 0.0001
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	true	false	0	15	690	0.0260 ± 0.0003
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	true	true	0	10	770	0.0264 ± 0.0003
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	false	false	0	0	3490	0.0383 ± 0.0005
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-5	false	true	0	5	3370	0.0381 ± 0.0005
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	true	false	0	15	690	0.0236 ± 0.0003
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	true	true	0	10	770	0.0247 ± 0.0003
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	false	false	0	0	3490	0.0238 ± 0.0003
10	2.5E-5	1.0E-4	2.0E-4	1.0E-5	1.0E-10	false	true	0	5	3370	0.0253 ± 0.0003

order 1). Configurations with 8 and 10 engines can tolerate double failures using pessimistic sensor defaults and non-segregated power wirings. Using pessimistic sensor defaults leads to an explosion of the number of minimal cutsets of order 3, which can increase unreliability if sensors are not sufficiently reliable. Indeed, a failing pessimistic sensor causes a spurious thrust reallocation, which leads to a trimming regime where engines fail more often. This results in a higher unreliability for the configurations that tolerate double failures. This tradeoff can be solved by increasing sensor reliability but this is to balance with cost aspects.

Listing 13 shows the ALPACAS code which generates the design-space of the system and selects the configuration without MCS of order 1 and with the lowest unreliability. The results can be further processed using the full Scala language, opening the door to design optimization taking into account other aspects such as the cost of the components, etc.

```
case class EngParams(nEng: Int, lam0: Double, lam1: Double, lam2: Double)

class ThrustRealloc(
  val engineParams: EngParams,
  val lambdaSensor: Double,
  val optimisticSensor: Boolean,
  val wiring : Wiring,
) extends Component { /* Model declaration */ }

val systems = for {
  eps      <- List(
    EngParams(6, 1E-5, 1E-4, 2E-4),
    EngParams(8, 2E-5, 1E-4, 2E-4),
    EngParams(10, 2.5E-5, 1E-4, 2E-4)
  )
  lamSens <- List(1E-5, 1E-10)
  optSens <- List(true, false)
  wiring  <- List(stdWiring(eps.nEng), segWiring(eps.nEng))
} yield ThrustRealloc(eps, lamSens, optSens, wiring)

var minUR = Double.PositiveInfinity
var bestSystem: Option[GenericPowertrain] = None
```

```
23
24 for {
25   system <- systems
26   model  <- stochasticCheck(system)
27   mcs    <- minimalCutSetsBFS(model, system.observer.isOk, 3)
28   urRes  <- unreliability(model, system.observer.isOk, 1E3, nbSimus, 8)
29   (_, urmax) = urRes
30 } {
31   if (mcs.forall(_.events.size > 1) && urmax < minUR) then
32     bestSystem = Some(system)
33     minUR = urmax
34 }
```

Listing 13 Design-space exploration example.

This study confirms that Alpacas is adapted to safety modeling of parametric families of architectures, and allows to compute safety indicators on the formalized design-space allowing to identify design tradeoffs and possibly to determine the optimal architecture with regard to a chosen metric (which might include other parameters than safety indicators, like cost).

8 Conclusion and Future Work

In this paper we presented Alpacas, a domain-specific language for system safety modeling and analysis. Using stochastic guarded transition systems as underlying formalism, it allows to model a large class of dynamic and re-configurable systems. It extends the state of the art in model-based safety assessment by bringing many cutting edge features from Scala 3 for generic programming thanks to a deep embedding. Parametric polymorphism, type-class polymorphism, higher-order parameters, higher-kinded types, etc. open the way to more efficient modeling and design-space formalization and exploration for safety critical systems. The Alpacas feature set was tested on a representative case study modeling a family of architectures for a thrust reallocation function for electric Vertical Takeoff and Landing aircraft. The scope of applications of Alpacas is not limited to aerospace systems and can benefit other domains such as automotive, railway, etc. which have similar safety processes [39, 46]. Alpacas is available under an academic open-source license on this repository `https://gitlab.com/maximebuyse/alpacas`.

The future work planned for Alpacas is the following. First, we will study how Scala 3's new macro system can improve the Monte-Carlo simulation performance, by inlining and specializing assertion, guards and transition evaluation functions, removing boxing as much as possible and distributing simulations on several computing cores. Second, we would like to connect this safety-oriented framework to existing Scala frameworks for temporal logic property monitoring such as DejaVu [27] or TraceContract [6]. This would allow to validate temporal logic properties on complex re-configurable system before deploying the temporal logic monitors for runtime safety assurance, and to derive process and reliability requirements for various autonomy functions. This would allow to monitor divergence between system models and actual system behaviour, and to trigger model updates to bridge the modeling gap. Third, we will study the connection of Alpacas to reinforcement learning frameworks, in order to study the synthesis of optimal policies for reconfiguration, repair and maintenance of complex critical systems.

References

1 Alexandre Albore, Silvano Dal Zilio, Guillaume Infantes, Christel Seguin, and Pierre Virelizier. A model-checking approach to analyse temporal failure propagation with altarica. In Marco Bozzano and Yiannis Papadopoulos, editors, *Model-Based Safety and Assessment*, pages 147–162, Cham, 2017. Springer International Publishing.

2 R. Alur and M. Bernadsky. Bounded model checking for GSMP models of stochastic real-time systems. In J.P. Hespanha and A. Tiwari, editors, *Hybrid Systems: Computation and Control, 9th International Workshop, HSCC 2006, Santa Barbara, CA, USA, March 29-31, 2006, Proceedings*, volume 3927 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 2006. doi:10.1007/11730637_5.

3 Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 249–272. Springer, 2016. doi:10.1007/978-3-319-30936-1_14.

4 André Arnold, Gérald Point, Alain Griffault, and Antoine Rauzy. The altarica formalism for describing concurrent systems. *Fundam. Informaticae*, 40(2-3):109–124, 1999. doi:10.3233/FI-1999-402302.

5 Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 1216–1225. ACM, 2012. doi:10.1145/2228360.2228584.

6 Howard Barringer and Klaus Havelund. Tracecontract: A scala DSL for trace analysis. In Michael J. Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *Lecture Notes in Computer Science*, pages 57–72. Springer, 2011. doi:10.1007/978-3-642-21437-0_7.

7 Pierre-Marie Basset, Binh Dang Vu, Philippe Beaumier, Gabriel Reboul, and Biel Ortun. Models and methods at onera for the presizing of evtol hybrid aircraft including analysis of failure scenarios. In *AHS Forum 2018, May 2018, PHOENIX, United States*, 2018.

8 Michel Batteux, Tatiana Prosvirnova, and Antoine Rauzy. System Structure Modeling Language (S2ML). preprint, 2015. URL: https://hal.archives-ouvertes.fr/hal-01234903.

9 Michel Batteux, Tatiana Prosvirnova, Antoine Rauzy, and Leïla Kloul. The altarica 3.0 project for model-based safety assessment. In *11th IEEE International Conference on Industrial Informatics, INDIN 2013, Bochum, Germany, July 29-31, 2013*, pages 741–746. IEEE, 2013. doi:10.1109/INDIN.2013.6622976.

10 Kevin Batz, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. Pric3: Property directed reachability for mdps. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 512–538. Springer, 2020. doi:10.1007/978-3-030-53291-8_27.

11 Pierre Bieber, Remi Delmas, and Christel Seguin. Dalculus - theory and tool for development assurance level allocation. In Francesco Flammini, Sandro Bologna, and Valeria Vittorini, editors, *Computer Safety, Reliability, and Security - 30th International Conference, SAFECOMP 2011, Naples, Italy, September 19-22, 2011. Proceedings*, volume 6894 of *Lecture Notes in Computer Science*, pages 43–56. Springer, 2011. doi:10.1007/978-3-642-24270-0_4.

12 Benjamin Bittner, Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Marco Gario, Alberto Griggio, Cristian Mattarei, Andrea Micheli, and Gianni Zampedri. The xsap safety analysis platform. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as*

Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, volume 9636 of *Lecture Notes in Computer Science*, pages 533–539. Springer, 2016. doi:10.1007/978-3-662-49674-9_31.

13 Marc Bouissou and Jean-Louis Bon. A new formalism that combines advantages of fault-trees and markov models: Boolean logic driven markov processes. *Reliab. Eng. Syst. Saf.*, 82(2):149–163, 2003. doi:10.1016/S0951-8320(03)00143-1.

14 Marco Bozzano, Alessandro Cimatti, Alberto Griggio, and Cristian Mattarei. Efficient anytime techniques for model-based safety analysis. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 603–621. Springer, 2015. doi:10.1007/978-3-319-21690-4_41.

15 Carlos E. Budde, Pedro R. D'Argenio, and Arnd Hartmanns. Automated compositional importance splitting. *Sci. Comput. Program.*, 174:90–108, 2019. doi:10.1016/j.scico.2019.01.006.

16 Pierre-Yves Chaux, Jean-Marc Roussel, Jean-Jacques Lesage, Gilles Deleuze, and Marc Bouissou. Systematic extraction of minimal cut sequences from a BDMP model. In *21st European Safety & Reliability Conference (ESREL 12), Jun 2012, Helsinki, Finland*, 2012.

17 Shengxin Dai, Mei Hong, and Bing Guo. A comparative study of reliability-ignorant and reliability-aware energy management schemes using UPPAAL-SMC. *Sci. Program.*, 2017:2621089:1–2621089:12, 2017. doi:10.1155/2017/2621089.

18 Patrick R. Darmstadt, Ralph Catanese, Allan Beiderman, Fernando Dones, Ephraim Chen, Mihir P. Mistry, Brian Babie, Mary Beckman, , and Robin Preator. Hazards analysis and failure modes and effects criticality analysis (fmeca) of four concept vehicle propulsion systems. Technical report, NASA/Boeing, 2019. URL: https://hummingbird.arc.nasa.gov/Publications/files/CR-2019-220217.pdf.

19 Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

20 Julien Delange and Peter H. Feiler. Architecture fault modeling with the AADL error-model annex. In *40th EUROMICRO Conference on Software Engineering and Advanced Applications, EUROMICRO-SEAA 2014, Verona, Italy, August 27-29, 2014*, pages 361–368. IEEE Computer Society, 2014. doi:10.1109/SEAA.2014.20.

21 Kevin Delmas, Rémi Delmas, and Claire Pagetti. Smt-based architecture modelling for safety assessment. In *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–8. IEEE, 2017.

22 Ewen Denney, Ganesh Pai, and Josef Pohl. Advocate: An assurance case automation toolset. In Frank Ortmeier and Peter Daniel, editors, *Computer Safety, Reliability, and Security - SAFECOMP 2012 Workshops: Sassur, ASCoMS, DESEC4LCCI, ERCIM/EWICS, IWDE, Magdeburg, Germany, September 25-28, 2012. Proceedings*, volume 7613 of *Lecture Notes in Computer Science*, pages 8–21. Springer, 2012. doi:10.1007/978-3-642-33675-1_2.

23 T. English and R. Heydor. Monte carlo simulation of markov, semi-markov, and generalized semi-markov processes in probabilistic risk assessment, final report. Nasa summer faculty fellowship program 2004, NASA, August 2005.

24 Majdi Ghadhab, Sebastian Junges, Joost-Pieter Katoen, Matthias Kuntz, and Matthias Volk. Safety analysis for vehicle guidance systems with dynamic fault trees. *Reliab. Eng. Syst. Saf.*, 186:37–50, 2019. doi:10.1016/j.ress.2019.02.005.

25 A. Hartmanns. MODEST - A unified language for quantitative models. In *Proceeding of the 2012 Forum on Specification and Design Languages, Vienna, Austria, September 18-20, 2012*, pages 44–51. IEEE, 2012. URL: http://ieeexplore.ieee.org/document/6336982/.

26 Klaus Havelund and Rajeev Joshi. Modeling with scala. In *International Symposium on Leveraging Applications of Formal Methods*, pages 184–205. Springer, 2018.

27 Klaus Havelund, Doron Peled, and Dogan Ulus. Dejavu: A monitoring tool for first-order temporal logic. In *3rd Workshop on Monitoring and Testing of Cyber-Physical Systems, MT@CPSWeek 2018, Porto, Portugal, April 10, 2018*, pages 12–13. IEEE, 2018. doi:10.1109/MT-CPS.2018.00013.

28 Paul Hudak. Building domain-specific embedded languages. *Acm computing surveys (csur)*, 28(4es):196–es, 1996.

29 Cyrille Jégourel, Axel Legay, and Sean Sedwards. Importance splitting for statistical model checking rare properties. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 576–591. Springer, 2013. doi:10.1007/978-3-642-39799-8_38.

30 Sohag Kabir. An overview of fault tree analysis and its application in model based dependability analysis. *Expert Systems with Applications*, 77:114–135, 2017.

31 Sohag Kabir, Yiannis Papadopoulos, Martin Walker, David Parker, Jose Ignacio Aizpurua, Jörg Lampe, and Erich Rüde. A model-based extension to hip-hops for dynamic fault propagation studies. In Marco Bozzano and Yiannis Papadopoulos, editors, *Model-Based Safety and Assessment - 5th International Symposium, IMBSA 2017, Trento, Italy, September 11-13, 2017, Proceedings*, volume 10437 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2017. doi:10.1007/978-3-319-64119-5_11.

32 Shahid Khan, Joost-Pieter Katoen, and Marc Bouissou. A compositional semantics for repairable bdmps. In António Casimiro, Frank Ortmeier, Friedemann Bitsch, and Pedro Ferreira, editors, *Computer Safety, Reliability, and Security - 39th International Conference, SAFECOMP 2020, Lisbon, Portugal, September 16-18, 2020, Proceedings*, volume 12234 of *Lecture Notes in Computer Science*, pages 82–98. Springer, 2020. doi:10.1007/978-3-030-54549-9_6.

33 Shahid Khan, Joost-Pieter Katoen, and Marc Bouissou. Explaining boolean-logic driven markov processes using gspns. In *16th European Dependable Computing Conference, EDCC 2020, Munich, Germany, September 7-10, 2020*, pages 119–126. IEEE, 2020. doi:10.1109/EDCC51268.2020.00028.

34 Oleg Kiselyov. Typed tagless final interpreters. In *Generic and Indexed Programming*, pages 130–174. Springer, 2012.

35 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: probabilistic model checking for performance and reliability analysis. *SIGMETRICS Perform. Evaluation Rev.*, 36(4):40–45, 2009. doi:10.1145/1530873.1530882.

36 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011. doi:10.1007/978-3-642-22110-1_47.

37 N. Limnios and G. Oprişan. *Semi-Markov Processes and Reliability*. Number 1 in Statistics for Industry and Technology. Birkhäuser, Boston, MA, 2001. doi:10.1007/978-1-4612-0161-8.

38 O. Lisagor, T. Kelly, and R. Niu. Model-based safety assessment: Review of the discipline and its challenges. In *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*, pages 625–632, 2011. doi:10.1109/ICRMS.2011.5979344.

39 Joseph Machrouh, Jean-Paul Blanquart, Philippe Baufreton, Jean-Louis Boulanger, Hervé Delseny, Jean Gassino, Gerard Ladier, Emmanuel Ledinot, Michel Leeman, Jean-Marc Astruc, Philippe Quéré, Bertrand Ricque, and Gilles Deleuze. Cross domain comparison of System Assurance. In *Embedded Real Time Software and Systems (ERTS2012)*, Toulouse, France, 2012. URL: https://hal.archives-ouvertes.fr/hal-02170444.

40 Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicitly: foundations and applications of implicit function types. *Proc. ACM Program. Lang.*, 2(POPL):42:1–42:29, 2018. doi:10.1145/3158130.

41 Antoine Rauzy. Binary decision diagrams for reliability studies. In *Handbook of performability engineering*, pages 381–396. Springer, 2008.

42 Antoine B Rauzy. Guarded transition systems: a new states/events formalism for reliability studies. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, 222(4):495–505, 2008.

43 Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-virtualized: linguistic reuse for deep embeddings. *High. Order Symb. Comput.*, 25(1):165–207, 2012. doi:10.1007/s10990-013-9096-9.

44 Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012. doi:10.1145/2184319.2184345.

45 Amir Shaikhha and Lionel Parreaux. Finally, a polymorphic linear algebra language (pearl). In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, volume 134 of *LIPIcs*, pages 25:1–25:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ECOOP.2019.25.

46 Ioannis Sorokos, Luis P. Azevedo, Yiannis Papadopoulos, Martin Walker, and David J. Parker. Comparing automatic allocation of safety integrity levels in the aerospace and automotive domains. *IFAC - PapersOnLine*, 49(3):184–190, 2016. 14th IFAC Symposium on Control in Transportation Systems 2016. doi:10.1016/j.ifacol.2016.07.031.

47 Alain Villemeur. *Reliability, availability, maintainability and safety assessment, assessment, hardware, software and human factors*, volume 2. Wiley, 1992.

48 Håkan L. S. Younes and Reid G. Simmons. Solving generalized semi-markov decision processes using continuous phase-type distributions. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 742–748. AAAI Press / The MIT Press, 2004. URL: http://www.aaai.org/Library/AAAI/2004/aaai04-117.php.

49 Tian Zhao and Xiaobing Huang. Design and implementation of deepdsl: A DSL for deep learning. *Comput. Lang. Syst. Struct.*, 54:39–70, 2018. doi:10.1016/j.cl.2018.04.004.

CodeDJ: Reproducible Queries over Large-Scale Software Repositories

Petr Maj[1]
Czech Technical University in Prague, Czech Republic

Konrad Siek[1]
Czech Technical University in Prague, Czech Republic

Alexander Kovalenko
Czech Technical University in Prague, Czech Republic

Jan Vitek
Czech Technical University in Prague, Czech Republic
Northeastern University, Boston, MA, USA

Abstract

Analyzing massive code bases is a staple of modern software engineering research – a welcome side-effect of the advent of large-scale software repositories such as GitHub. Selecting which projects one should analyze is a labor-intensive process, and a process that can lead to biased results if the selection is not representative of the population of interest. One issue faced by researchers is that the interface exposed by software repositories only allows the most basic of queries. CodeDJ is an infrastructure for querying repositories composed of a persistent datastore, constantly updated with data acquired from GitHub, and an in-memory database with a Rust query interface. CodeDJ supports reproducibility, historical queries are answered deterministically using past states of the datastore; thus researchers can reproduce published results. To illustrate the benefits of CodeDJ, we identify biases in the data of a published study and, by repeating the analysis with new data, we demonstrate that the study's conclusions were sensitive to the choice of projects.

2012 ACM Subject Classification Software and its engineering → Ultra-large-scale systems

Keywords and phrases Software, Mining Code Repositories, Source Code Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.6

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.13`

Funding This work is supported by the Czech Ministry of Education, Youth and Sports from the Czech Operational Programme Research, Development, and Education, under grant agreement No.CZ.02.1.01/0.0/0.0/15_003/0000421 and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 695412).

1 Introduction

With over 190 million public projects, GitHub is our largest source of empirical data about how software is developed. It is a treasure trove that must be mined if we want to distill insights from its contents. Manual inspection is limited to small-scale case studies; even automated analysis tools struggle with the sheer amount of data available. The software engineering community has taken up this challenge, researchers examine increasingly larger numbers of projects in order to test hypotheses and derive knowledge about the software development process. Examples of such studies include investigations of testing practices [12], changes to licensing over time [18], popularity trends [4] and configuration settings [17].

[1] These authors contributed equally.

These works use samples of GitHub ranging from 15K to 100K projects filtered to exclude projects considered as lacking in size, popularity, originality or importance.

For any scientific study of software, selecting the projects that make up the input of that study is fraught with risks. Any given choice can introduce unwanted and sometimes undetected bias. This bias may, in turn, taint the conclusions of the work. Much like the task of polling voters before an election, choosing a subset of a larger population must be done carefully. In polls, the goal is to ensure appropriate representation of likely voters. The chosen subset excludes citizens who are either not eligible or unlikely to vote, and balances the various population groups. At the same time, for reasons of cost and practicality, the size of this subset is kept as small as possible. Even when pollsters are careful, the accuracy of predictions varies. In software engineering, we often look for some properties of "real" code – where our definition of the term is sensitive to context and research goals. One may exclude course assignments because the errors made by beginners are not relevant to deployed software; on the other hand, if our goal is to shine a light on acquisition of programming skills, then that kind of code may be exactly what is needed. Picking the right set of inputs is thus the first challenge any researcher in the field must address.

With software, Nagappan et al. warned us that more is not always better [14]. Their observations hold now more so than back in 2013 as anyone can create a GitHub repository at no cost and house almost anything there. Manual inspection found that 37% of hosted projects are not used for software development [11]. Thus, the quality of data gathered from software repositories should always be questioned. A stark illustration why skepticism is in order comes from the finding that ten common source corpora have up to 68% of bit-for-bit identical file duplicates [1]. Furthermore, the same paper showed that clones impacted the accuracy of results obtained with these corpora. We argue that more is worse: as the number of projects to scrutinize grows, it becomes harder to check whether their data is clean, consistent and well-formed. Consider the case of text files accidentally misidentified as code [15], an error that went unnoticed for three years and was "fixed" by partially invalidating the original paper's conclusions [2]. As a result of this state of affairs, researchers spend significant effort collecting and curating meaningful suites of open source projects. Unfortunately, manual curation cannot track the constantly changing software landscape.

In this paper, we aim to address a seemingly simple yet eminently practical question, *how does one find software projects in large-scale software repositories?* The assumption underlying our work, our hypothesis, is that it is possible to select thousands of projects from millions by formulating queries on attributes found in the projects' metadata and on easily computed properties of their source code. To be concrete about the kinds of queries we envision, consider looking for the one hundred most popular projects predominantly written in Java, developed in the five years before the introduction of Lambdas by at least two developers with five years of experience. Furthermore, let's ensure that the selected projects have no more than 5% duplicate files between each other. While the search interface provided by software repositories may allow to query for projects by language, there is no way to compute this query automatically without retrieving all projects.

This paper reports on the status of CodeDJ, an infrastructure for querying large-scale software repositories. In its current incarnation our system is geared towards processing data from any git-based software repository. For our experiments, we specifically target GitHub. The three main engineering challenges we contend with are the sheer size of the data source, the constant updates to its data, and the narrow, rate-limited, interface for accessing projects. In addition, a key design requirement is reproducibility; not only should queries execute deterministically, but the infrastructure should be able to replay a historical query with identical results. Thus, researchers may take any query from the literature, even

years after it was originally run and its output was used in a publication, and match its results. Furthermore, researchers should be able to modify a historical query and run it based on the information available at any point in the past.

To address these challenges and requirements, CodeDJ is architected in two distinct subsystems. Interaction with the data source is mediated by Parasite, a time-indexed datastore that automatically and continuously queries GitHub for data about projects. Parasite is responsible for data acquisition and keeping that data up-to-date over time. Every datum is logically time-stamped to enable reproducibility. To ensure that CodeDJ can scale, Parasite can be split up into multiple distinct substores based on the projects' main language. The second subsystem, an in-memory database named Djanco, handles user-written queries. For each query, Djanco determines the portion of the datastore that is required, loads the data, and executes the query. Queries evaluate with project metadata in memory while source code remains on disk. The query syntax is based on data frame manipulation interfaces popular in data science, such as dplyr [19], and is expressed in Rust. We claim the following contributions:

- The design of CodeDJ, a scalable infrastructure for querying large-scale software repositories that supports reproducibility and continuously updated data sources.
- A prototype implementation of Parasite and Djanco written in Rust that shows scalability to millions of projects.
- A dataset consisting of 3.6 million software projects written in 17 languages obtained from GitHub.
- A case study illustrating that the choice of projects can invalidate the conclusion of a research project.

Equally important is what we don't do. We do not provide guidance on how to use our infrastructure. The determination of what is the *right* input for a given analysis is problem specific and the choice remains something individual researchers must grapple with. We have not shown scalability of our infrastructure to the whole of GitHub, we are comfortable with datastores of up to 10 million projects. A larger size may require more work. We do not support interactive queries, our infrastructure was designed with the understanding that queries can take hours to run. We did not focus on optimizing query evaluation by, e.g. parallelizing their execution. Lastly, we do not index any artifacts other than code. Adding images, configuration files and documentation is possible but was not considered one of our targets.

Availability. CodeDJ is an open source infrastructure. Readers interested in repeatability, will find our reproduction package at:

`https://github.com/PRL-PRG/codedj-ecoop-artifact`

The source code of Parasite and Djanco are on GitHub at:

`https://github.com/PRL-PRG/codedj-parasite`
`https://github.com/PRL-PRG/djanco`

As our datastore is too large to easily share, Sec. 3.3.4 discusses how external users can run queries on our servers. Another alternative is for users to set up their own CodeDJ instance and gather their own data to execute queries. Our reproduction package contains a complete walk-through of the set up procedure. Of course, users must publish their dataset to enable reproducibility.

2 Related Work

Table 1 gives a high-level comparison with eight systems with aims similar to ours. The first column (*Active*) indicates if the system is actively maintained. Some research projects have fallen into disrepair and their web pages are unreachable. The second column (*Updated*) indicates if continuous updates are supported. Given the rate of addition to GitHub, most systems struggle to keep up. The third column (*Reproducible*) indicates if results are reproducible. Reproducibility is only relevant when the data is updated, systems built on a single static snapshot trivially support reproducibility. The fourth column (*Consistent*) indicates that the data is consistent. Inconsistencies arise when some earlier data (such as parent commits) are missed. The fifth column (*Queries*) describes the nature of the query interface exposed to users. Some systems have a simple filtering mechanism for a fixed set of attributes, such as the language of the project, others have their own query language. In our case, we express queries in Rust. The sixth column (*Sources*) indicates where the data comes from. Mostly this is GitHub, but the Apache Software Foundation and various other sources have also been used in the past. The seventh column (*Size*) is an estimate of how many projects are available. Finally the last column (*Contents*) indicates if source code can be queried. Most systems only include metadata about projects due to the size of the code.

Table 1 Systems comparison.

	Active	Updated	Reproducible	Consistent	Queries	Sources	Size	Contents
Stress [8]	–	–	Y	Y	Filter	Apache	211	–
Flossmetrics [9]	–	–	Y	Y	Filter	Many	2.8**K**	–
Orion [3]	–	–	Y	Y	Own	Many	185**K**	Y
Boa [7]	Y	–	Y	Y	Own	Java	380**K**	Y
Black Duck	Y	Y	–	Y	Filter	Many	680**K**	–
Sourcerer [16]	–	–	Y	Y	Filter	GitHub	4.5**M**	–
GHTorrent [10]	Y	Y	–	Y	SQL	GitHub	157**M**	–
GitHub	Y	Y	–	–	Filter	GitHub	190**M**	Y
CodeDJ	Y	Y	Y	Y	Rust	GitHub	3.6**M**	Y

Stress. This system aims to help choose projects in a reproducible manner [8]. Its corpus consists of 211 projects which can be filtered on 100 pre-computed attributes such as bug tickets or lifetime. The corpus can be sorted and sampled randomly. Queries can be exported so they can be repeated later. Source code is not available for querying. Stress is inactive. CodeDJ scales to larger corpora and allows to specify richer queries. In terms of reproducibility, we support updates to the corpus.

Flossmetrics. This work analyzed 2800 open source projects and computed statistics about various aspects of their development process, such as number of commits and developers [9]. Information from additional sources such as project mailing lists and issue trackers was included. Queries could be formulated on metrics such as COCOMO effort, core team members, evolution and dynamics of bugs. Filtering based on these criteria was supported. The project is inactive and it did not support updates.

Orion. This system aimed to enable retrieving projects using complex search queries linking different artifacts of software development, such as source code, version control metadata, bug tracker tickets, developer activities and interactions extracted from the hosting platform [3]. The project is no longer maintained, it scaled to about 185K projects. CodeDJ is designed to scale to larger corpora and offers a more flexible query interface.

Boa. This system focuses on semantics queries over Java programs [7]. A corpus of 380K Java projects can be queried using a dedicated query language that supports automatic parallelization and pluggable mining functions. Source code can be queried in sophisticated ways as Boa is able to parse and analyze Java. A larger corpus of 7.5M projects can be queried on project summaries. Boa provides reproducibility by ensuring its queries are deterministic with respect to the dataset's version, which are created and archived infrequently (i.e. 2013, 2015, 2019, 2020). CodeDJ differs from Boa in that it is language agnostic and geared towards project selection, as opposed to project analysis. Furthermore, CodeDJ provides full reproducibility in the presence of a continuously evolving dataset.

Black Duck Open Hub. A public directory of open source software[2] that offers search services for discovering, evaluating, tracking, and comparing projects. It analyzes both the code's history and ongoing updates to provide reports about the composition and activity of code bases. CodeDJ allows researchers to write their own queries and supports reproducibility.

SourcererCC. The aim of this project is to detect code clones [16]. The tool scales to large datasets and can detect near-identical code at various granularities. It has been used to analyze cloning across large corpora of Java, JavaScript, Python, C and C++ projects on GitHub [13]. It can be used by researchers to detect duplication in their samples which is a source of bias. The project's web page appears to be inactive.

GHTorrent. This database of metadata about GitHub projects offers an SQL interface for queries [10]. It monitors GitHub events to constantly update the available data. The limitation of the approach is that GitHub's events do not have all commit details and file contents, thus these are not stored by GHTorrent. In our experience, the database is not always consistent, this may be due to missed events. We have attempted to upload queries through the public SQL interface but the queries timed out.

GitHub. This service provides two ways to query metadata and contents. A REST API can be used for requesting information about projects and listing them, its search queries provide filtering capabilities across a small set of fixed attributes. A web API provides extended filtering options such as searching within repositories written in a particular language. These interfaces are rate-limited and thus return partial results. The results are non-deterministic and non-reproducible as projects may be added and deleted at any time. CodeDJ provides a view of a subset of GitHub on which we support reproducibility and our queries are richer and deterministic.

We would be remiss if we failed to mention the Software Heritage Archive which aims to preserve all publicly available source code; currently upwards of 9.5B source files, 2B commits and 150M projects [6]. It only allows retrieval of single objects. The authors point to the fragility of current arrangements and the dynamic nature of source code repositories

[2] `https://www.openhub.net`

makes it difficult to reproduce studies that use them. We have encountered this ourselves: we see projects deleted from GitHub, changing names, or visibility. In the future, CodeDJ can be extended to query the heritage corpus as well as other repositories.

3 An Infrastructure for Querying Large-Scale Repositories

The goal of CodeDJ is to allow researchers to formulate queries that evaluate attributes of projects hosted on GitHub and return data about projects matching a specified predicate.

3.1 Design considerations and system architecture

The design of CodeDJ flows from four high-level principles that we motivate next:

- **Consistent, eventually:** The sheer size and churn in data sources such as GitHub means that obtaining a snapshot of the whole data source is not practical. But, it is often the case that a slightly out-of-date view is sufficient for most investigations. We choose to refresh entire projects atomically at irregular intervals. Thus, any individual project is consistent, but for any group of projects, the lower bound on their refresh times is the last consistent time point (`git` histories can be destructively updated, allowing for post factum inconsistencies, we ignore these).
- **Code-centric, language agnostic:** We aim to support queries on project metadata and file contents written in any programming language. To reduce space requirements, the only source artifacts we store is code, deduplication is used to remove redundancy, and metadata is trimmed where possible.
- **Flexible query interface:** Popular data science tools such as dplyr [19] or Spark [20] offer a mix operations inspired by database query languages extended with general purpose capabilities. Inspired by these, we propose an interface expressed in Rust as a library with operations for selecting, grouping, filtering and sampling data. The benefits of our approach over, say, SQL, is that queries are type-safe and benefit from the full generality of the Rust language.
- **Reproducible by design:** The importance of reproducibility cannot be overstated [5], consider [15] which recorded the names of the most starred projects seven years ago, without author names it is not possible uniquely to identify projects, and even with their full names, reconstructing a historical star count is not possible. CodeDJ is designed so it is possible to run any query with the information that the datastore had at an arbitrary point in the past. For this purpose the datastore is time-indexed, strictly append-only.

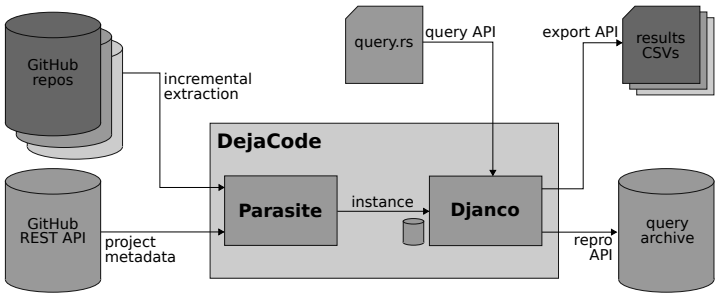


Figure 1 System overview.

Fig. 1 overviews the architecture of CodeDJ. The system is structured around two components, Parasite, a datastore that tracks GitHub, and Djanco, an in-memory database with a Rust

interface. Parasite is set up to continuously extract information from GitHub using its REST API for some data and cloning project repositories for other data. The information obtained from the data source is deduplicated and stored in a dedicated format on disk. At irregular intervals projects are refreshed, and the new information is appended to the datastore. When an end-user query is submitted for execution, it comes as a Rust function calling the Djanco query API, a database instance is created for that query. The database will load the data needed for query execution from Parasite. The output of a query is some results, usually as a text file and a record of that query in a reproducibility archive.

The remainder of this section describes our implementation, the design of the query interface and our support for reproducibility.

3.2 The Parasite datastore

Parasite is a dedicated, perpetually running application whose task is to synchronize its on-disk representation with GitHub. This task is complicated by these four constraints:

- **Scalable:** We expect to grow to hundreds of millions of projects, the disk format must be space efficient and its in memory format must be compact and fast to access.
- **Peaceful co-existence:** We must abide by GitHub's terms of service. Parasite must be economical in both the number requests to the GitHub API calls and raw `git` operations.
- **Time-indexed:** Every datum in the store must be associated with its acquisition date, this feature must have a minimal overhead so as not to increase our footprint.
- **Robust:** Backups are not possible due to limited resources, the datastore must thus be resilient to corruption.

Our description focuses on three aspects, the data acquisition process, the data storage format and the interface exposed to Djanco. We also explain how we meet the above constraints.

3.2.1 Acquisition

While, in theory, the GitHub API is sufficient to fulfill all our needs, the fact that GitHub defends itself against denial of service attacks limiting users to 5,000 requests per hour causes a practical problem. As every commit requires one request, the interface is too restrictive to collect data within a reasonable amount of time. Therefore, instead of relying on the API alone, Parasite combines a number of interfaces:

- **Git:** we use the `git clone` command to retrieve source code files and commit histories from repositories;
- **GitHub:** we use the REST API for project metadata (stars, watchers, issues, etc.), information that cannot be obtained through `git` alone;
- **GHTorrent:** instead of querying GitHub for projects directly, we seeded Parasite with the URLs of projects obtained from GHTorrent.[3]

Parasite continuously downloads data from its data sources on a per-project basis. The projects known to Parasite are maintained in a priority queue. Projects are visited in inverse order of last access time. Thus, given any group of projects, the lower bound on the time they were last visited determines the last point when Parasite had a consistent view of those projects modulo destructive `git` history rewrites.

[3] While GHTorrent has over 100M URLs, they are not all valid. Out of 5.5M URLs we visited, only 3.6M were usable, the remaining are either duplicates, have been deleted, or have become private.

When a project is visited, the download procedure begins. First, the project's metadata is retrieved via a call to the REST API. This yields a JSON file with metadata and sundry information. The metadata is stripped of non-essential information (such as URLs for various REST API requests) and stored. The project's current and last known URLs are compared to detect renaming and the new URL is recorded if a change occurred. Next, the project's heads are checked against the heads in the datastore. Each head corresponds to a branch in `git`. If any of the heads changed, the project is cloned and data about new commits and the contents of changed files are extracted and stored. We clone projects because using the REST API to get new commits is slow and rate limited. We clone repeatedly at each visit, caching projects is not feasible due to space limitations (in the future, we plan to cache the most active projects to reduce the amount of data unnecessarily transferred via full clones).

Once a local copy of a project exists, we determine which *substore* that project belongs to and append new commits and files to it. Substores are partitions of the dataset that Parasite uses to organize its disk structures around. Projects are matched to a single substore by properties such as size (e.g. a substore for small projects) or dominant language (e.g. a substore of Python projects).

When processing a chain of commits, a simple optimization is achieved by observing that if we find a commit that is already in the datastore, then all of its parent commits must also already be present. The final step is to record the time of the visit, and move to next project in the queue. Any error during the processing, terminates the visit and the project is flagged as potentially invalid.

Parasite is written in Rust using `libgit2`. It has been parallelized at project-level granularity and scales up to 32 threads. With more threads, the bottleneck shifts from local repository analysis to network bandwidth and ultimately to the GitHub rate limit. When adding projects, Parasite processes 244 projects per thread per hour. As GitHub limits are attached to users (identified by tokens), Parasite supports rotating multiple tokens which allow us to sustain a download rate of 7821 projects per hour using 32 threads. Since Parasite is still in accretion mode, we cannot report on the update rate alone, but we expect it to be limited by GitHub to a rate of 120K active project updates per day per token.

Table 2 Current dataset composition.

	Records	Size	Ratio
Users	4.8M	200M	<0.01%
Projects	3.6M	4.9G	0.2%
Commits	167M	88G	3.2%
Paths	848M	80G	2.9%
Files	463M	2603G	93.7%

Parasite has visited 3.6M projects composed from all non-fork C++ and Python projects available in GHTorrent and a random subset of 50K projects in 17 popular languages. In total, the datastore has 3.6M projects and occupies 2.8TB on disk. Table 2 shows that the majority of the datastore is taken by source code.

3.2.2 Storage

The storage format of Parasite is designed to ensure a low disk footprint, to scale to hundreds of millions of projects. The store is append-only to allow reverting to historic states and to simplify recovery from data corruption. Parasite can be thought of as storing *records*.

Records of same kind are backed by a single *record file*. Records compose together to form *entities*. The following entities are stored by Parasite:

- **Projects:** A project is identified by unique `git` clone URL, it has a set of *heads* (one per branch) and other information from GitHub metadata.
- **Commits:** A commit is identified by its SHA hash, it has a message, changes, parents, an author, a committer, and a time.
- **Paths:** A path is identified by the hash of its string value.
- **Users:** A user is identified by their email.
- **Snapshots:** A snapshot of a file containing source code is identified by its hash.

Records are the smallest unit of information in the datastore, the only way to update an entity is to add a new record. The decomposition of entities to records has been designed along the lines of what information can be updated in isolation. Entities are assigned unique numeric *identifiers* based on their contents. One of the key internal data structures in Parasite are the multiple *mappings* from entity hashes to identifiers. These mappings are used for deduplication.

Deduplication is crucial as up to 94% of files can be duplicates [13]. Mappings are costly as they must be kept in memory. For our corpus, the deduplication mappings for all entities require 89GB. While not a concern at this time, as our dataset grows, mappings will become a bottleneck. To decrease their size, we split Parasite into *substores*. Each substore manages a disjoint partition of the projects. We perform deduplication only within substores. This means that mappings are smaller at the price of some duplication across substores. Our implementation assigns projects to substores based on their size and dominant language; small projects (less than 10 commits) are kept distinct from projects written in targeted languages. A drawback of this design is that identifiers are not unique, if multiple substores must be accessed, extra care must be taken when merging their contents. On the other hand, this compartmentalization has immediate benefits: In terms of robustness, different substores can be stored in different locations and a loss of one does not impact the others. In terms of performance, queries can trivially skip reading irrelevant substores. We measured the duplication across substores at only 5.1%.

As source code (snapshots) dominate the datastore, Parasite internally splits snapshots by language, storing each language separately. This improves reading times for queries that filter by language.

Parasite avoids storing information that is expensive to update and that can be computed readily. For instance, the relation between commits and their project is not stored; it can be recovered from project heads and commit parents. To further reduce footprint, larger records are compressed. For snapshots, the compression ratio is 70%.

To quickly find the latest records for a particular entity, Parasite computes indices, which are stored in dedicated *index files* that provide, for each entity, the location of the latest version of its constituent records. These index files are updated in place as new records are added which exposes them to the risk of being inconsistent. If this occurs, they can always be recomputed from scratch. As of this writing, all indices in the datastore comprised 0.6% of our disk footprint.

To ensure that it is possible to associate a time with every datum on disk, Parasite introduces the notion of a *savepoint*. Since the store is append-only, time-indexing in Parasite boils down to simply associating a time to the current position of each substore. For consistency, savepoints can only be created between visits of projects. They are thus both a mechanism for reproducibility and robustness. Any query can be re-executed at any savepoint and will see the same information. The datastore can be rolled back to a savepoint in case of data corruption.

3.2.3 Interfaces

Parasite has two interfaces, one for data acquisition and another for reading data.

For monitoring purposes data acquisition exposes a detailed breakdown of running tasks, their progress and the usage of GitHub resources. Parasite has both an interactive text-based interface and a command-line interface for automation via scripts. These interfaces allow to create savepoints, verify integrity of the datastore and repair data corruption by reverting to previous savepoints. Parasite monitors available memory to keep as many mappings in memory as it can. Most of the datastore management can be done without reloading any mappings; the initial load takes 26 minutes.

The read interface allows to access records. Iterators are created relative to a savepoint and return records in the order they were added up to that savepoint. Many records are never superseded, for these iterator return values can be used as such. For records that can be overridden with newer values, iterators return updates in reverse chronological order. For projects, Parasite assembles their information; this takes some time as URLs, heads, update status, substore, and metadata must be loaded first, assembly discards all but the most recent versions. Iterators are geared towards sequential access to all elements, but the index files kept by Parasite can be used for random access as well.

3.3 The Djanco database

The Djanco database acts as an intermediary between Parasite and the end-user. It provides a robust query engine that manages loading and pre-processing data and a domain-specific language to express queries easily and concisely. Finally, it supports replaying historical queries. Djanco is designed under the following simplifying assumptions:

- **Single-user:** Djanco is used by a single user for a single query at a time; any parallelism is internal and transparent.
- **Determinism:** Queries are fully replayable on the basis of parameters explicitly provided by the end-users such as random seeds, timestamps, and data source.
- **Read-only:** Queries cannot update the datastore, changes are limited to local objects and are not persisted.
- **Fixed-schema:** Djanco only contains data and metadata pertaining to GitHub.

The need for Djanco comes from the structure of Parasite. The datastore is designed to allow continuous updates and to decrease footprint. This complicates answering research questions. For instance, Parasite elides the relation from a project to its commits. A simple question such as how many commits there are in a project requires recomputing that relation by looking up one the of project's branches and its most recent commit. From that commit, one can follow the parent commits and recursively enumerate them all. Then, repeat for all branches. The database layer computes relations such as these and caches data persistently to speed up queries.

The rationale for a dedicated database rather than an off-the-shelf one are threefold. First, and most arguably, our experience using MySQL on a related project suggested that scalability to large data size (2.8TB and growing) can lead to significant execution overheads. Secondly, we can leverage the assumptions above to implement a domain-specific database as many features of traditional databases (transactions, locks, a general schema) are superfluous. Instead, we implement a solution specialized to our schema that lazily loads selected data from the datastore. Finally, some of our queries are difficult to express in the relational model. Queries can become lengthy and involve multiple joins, nesting and views, which makes them difficult to debug and maintain.

3.3.1 Instances

A Djanco database *instance* is logically created for each end-user query. Each instance is irrevocably tied to a specific slice of the datastore. This slice is defined by two parameters: the substores that indicate which projects to load, and a timestamp indicating a savepoint to be checked out from each substore. If multiple datastores are used, the database joins and deduplicates them. The Djanco schema is shown in Fig. 2, it defines five different

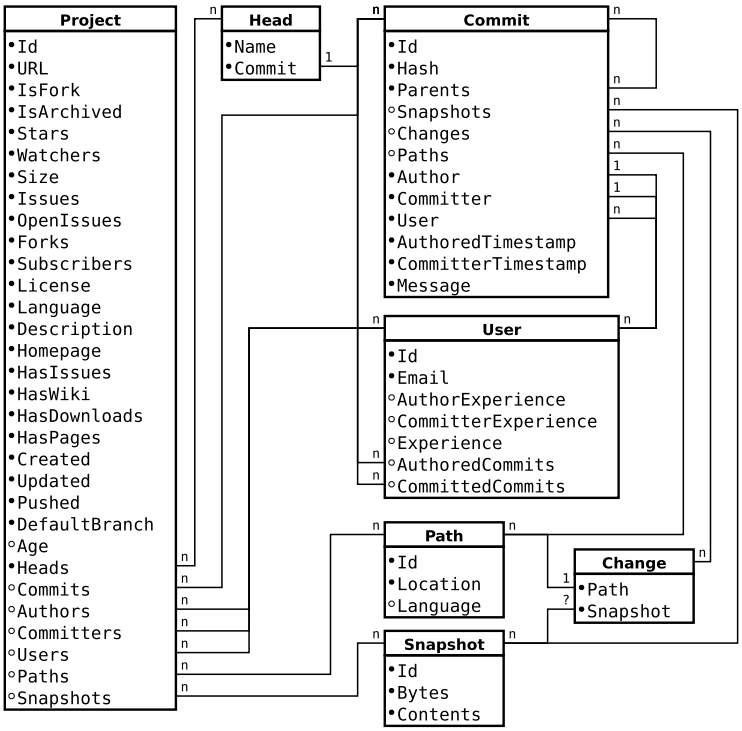


Figure 2 Djanco schema (computed attributes marked ◦).

entities: projects, commits, paths, users and snapshots. Each with their own attributes and convenience methods. Even though Djanco derives its schema from Parasite, there is not a one-to-one correspondence between them. While Parasite tends towards generality and frugality, Djanco instead tends towards expressivity and convenience. For instance, Parasite stores project metadata in JSON, while Djanco parses the format, extracts useful information at sensible types. The basic information about projects is their ID and URL. The metadata includes:

- the language as determined by GitHub;
- the numbers of stars, watchers, subscribers, issues, and forks;
- dates for creation, most recent update, and most recent push;
- the license, description, and homepage URL;
- which web services are active: issues, wiki, downloads, pages;
- size in bytes;
- name of the default branch (e.g. “master” or “main”);
- whether the project is archived or a fork.

Djanco provides a method to calculate the age of a project as the span of the time between its first and most recent commit. Finally, it provides methods to retrieve relations between a

project and other entities: heads, commits, users, authors, committers, paths, and snapshots. Except for heads, all the relations need to be computed.

Commits have IDs, hashes, messages, as well as timestamps at which they were authored and pushed. Each commit is associated with users, having an author and a committer. A commit also has a list of changes: a change is a modification to a file represented by a path in the repository and the contents of the file after the change. Finally, commits reference a list of zero or more parent commits in the commit tree.

Users have IDs and emails. In addition, experience is computed for authors and committers as the timespan between their first and last commit. Users also have a method to acquire the list of commits they authored or committed.

Paths represent file system locations within the project (e.g. `"src/main.c"`). They are identified by a synthetic ID and contain a string representing the path. A method to guess the language of a file from its extension is provided. Snapshots are the stream of bytes that are contents of a file at some point in time. For instance, if a file is edited during a commit, the contents of that file before and after the edit are two separate snapshots.

3.3.2 Queries

Queries can be expressed either through a low-level interface or via a DSL. The former accesses the schema directly with Rust iterators and methods. The DSL is a more compact way to implement common queries.

The first step for all queries is to construct a database instance. Since an instance wraps around a specific view of the datastore, constructing it requires specifying a path, a savepoint and substores. The following snippet constructs an instance for small projects available on December 1st, 2016:

```
let db = Django::new(PATH, timestamp!(December 2016), substore!(SmallProjects))?;
```

Alternatively, an instance for C, C++, and Python programs is constructed like this:

```
let db = Django::new(PATH, timestamp!(December 2016), substores!(C, C++, Python))?;
```

Parameters can be skipped; an instance from all substores at their most recent savepoint is constructed thus (values of defaults are recorded for reproducibility):

```
let db = Django::from(PATH)?;
```

Iterators offer access to entities. The snapshot iterator is lazy, the others eagerly load information from the datastore. Iterators are entry points to queries; they return objects that conform to the schema of Fig. 2. This snippet extracts a vector of all languages occurring in projects:

```
let all_languages = db.projects()
  .map(|project| project.language())
  .unique()
  .collect()::<Vec<Language>>;
```

While iterators suffice for just about any query, most queries can be expressed more concisely in our DSL. The DSL uses a pipeline paradigm, where an initial data structure is transformed by a series of methods (aka verbs) that do part of the processing in each step. We provide the following verbs: `group`, `filter`, `sort_by`, `sample`, and `map_into`. We also provide access to any attribute in the schema. In addition, objects and their attributes are composable into complex statements expressing comparisons (e.g. `AtLeast`, `AtMost`, `Matches`, `Contains`), basic statistical functions (`Count`, `Max`, `Median`), sampling methods (`Top`, `Random`), and many others. The code below showcases a few of these:

```
let selection = db.projects()
  .group_by(project::Language)
  .filter_by(AtLeast(Count(project::Users), 5))
  .sort_by(project::Stars)
  .sample(Top(50));
```

Projects are grouped according to their language, then filtered so that only projects that have at least 5 users are kept, these are sorted by the number of stars in each project and, finally, a sample of top 50 projects is returned.

A useful feature is the ability to deduplicate projects while sampling them according to specific criteria. For example, in the following snippet projects will not be added to the result set unless 90% of their commits are unique with respect to any other project already within the result set:

```
selection.sample(Distinct(Top(50), MinRatio(project::Commits, 0.9)))
```

The final step of a query is to output its results; here we show results written to a CSV file:

```
selection.into_csv(OUTPUT_PATH)?;
```

Each object serializes verbosely, including all information about itself. If only specific information is required, an appropriate format may be imposed by using the map verb to translate an object into its attributes. Here each project is translated into its ID and URL:

```
selection
  .map_into(Select!(project::Id, project::URL))
  .into_csv(OUTPUT_PATH)?;
```

We also provide a function that outputs all information related to a project, including commits, users, paths and snapshots. This creates multiple CSV files.

```
selection.dump_all_info_to(OUTPUT_DIR_PATH)?;
```

Crucially, end-users can do their own use-case–specific formatting by resorting to Rust:

```
selection.for_each(|project| println!("{}:␣{}", project.url(), project.has_wiki()))
```

Further details about our query facilities can be found in the Djanco GitHub repository.

A friend in need. We had an opportunity to test our system when posed a question that was difficult to answer with GitHub's REST API. The query had to retrieve popular C++ repositories that use custom allocators. Finding out whether a project is using a custom allocator requires checking if it imports a library called `memory_resource`. Therefore, we

```
let wanted: HashMap<SnapshotId> = db
 .snapshots()
 .filter(|snapshot|
   snapshot.contains(
    "#include␣<memory_resource>"))
 .map(|snapshot| snapshot.id())
 .collect();

let projects = db.projects()
 .filter(|project| {
  project.snapshots()
   .map_or(false, |snapshots| {
    snapshots.iter()
     .map(|snapshot| snapshot.id())
     .any(|snapshot_id| {
      wanted.contains(snapshot_id)
     })
   })
 .sorted_by_key(|project|
  project.star_count());
```

```
let wanted: HashSet<SnapshotId> = db
 .snapshots()
 .filter_by(
   Contains(snapshot::Contents,
      "#include␣<memory_resource>"))
 .map_into(snapshot::Id)
 .collect();

let projects = db.projects()
 .filter_by(
   AnyIn(project::SnapshotIds, wanted))
 .sort_by(project::Stars);
```

Figure 3 Emery query.

`grep` through source code for the string `"#include␣<memory_resource>"`. In a second step, we iterate over projects and find those, which contain one of the selected snapshots. At that point, we order them by popularity and retrieve some number of the most popular projects. For comparison we wrote the query in pure Rust and then in the DSL. Both implementations are in Fig. 3. As expected the DSL is more compact and more readable. We ran the query on a store with 3M projects and 429M snapshots. The first part of the query found 1724 snapshots in 12 hours. The second part of the query retrieved 1197 projects and their metadata in 24 hours. Then, an additional 6 hours was spent on preparing the project metadata for CSV export.

3.3.3 Data management

Djanco transparently manages the loading and pre-processing of data from the datastore. This involves two mechanisms: lazy loading and caching. Given the size of the data, loading it all into memory is not desirable. Most queries are interested with a small slice of the data, usually filtering out most projects and neglecting most attributes. Therefore, Djanco uses lazy loading to tailor the in-memory data to the needs of each specific query. Snapshots (source code files) are bulky and cannot be split into independent attributes. Only a single snapshot is held in memory at once. The database retrieves them from the datastore only when needed either by scanning the store sequentially or by using the datastore's ability to seek and access a specific snapshot. For the other objects (projects, commits, paths, and users), their attributes are loaded independently on request. Attributes are cached in the database as they can be needed several times.

Memory usage is not the only concern while loading data from the store. From our experiences in querying GitHub, we find that many similar queries are executed on the same datastore view, especially when a query is being developed. Loading attributes from the datastore can be costly, especially in places where the Djanco schema requires the values to be calculated, e.g. for mappings between entities. Therefore, we found it beneficial to avoid recalculating some attributes across queries by implementing on-disk attribute caching, thus improving performance of similar or repeated queries.

For each attribute occuring in a query, the database creates an in-memory map, mapping an entity ID to that entity's value for a given attribute. After an attribute has been loaded, the caching extension serializes it onto disk using the CBOR serialization format. The on-disk cache structure preserves information about which datastore, savepoint, and substore a particular attribute map was read from. Subsequent queries requesting this attribute for this particular datastore view then prefer loading data from the cache rather than the datastore. This process is transparent to the end-user, and can be turned off to save disk space.

Table 3 Caching performance.

	extracting from store	writing to cache	reading from cache	size on disk	cached?
`commit::Parents+commit::Users`	1h 21m 28s	35m 16s	7m 25s	2.3GB	Y
`user::Experience`	1h 10m 19s	1s	1s	5.7MB	Y
`user::CommitterExperience`	1h 9m 52s	1s	1s	5.6MB	Y
`user::AuthoredCommits`	1h 8m 47s	1m 1s	39s	213MB	Y
`project::Commits`	1h 8m 33s	5m 29s	3m 25s	1.1GB	Y
`commit::Changes`	52m 29s	2h 53m 53s	1h 21m 28s	20GB	N
`commit::CommitterTimestamp`	41m 49s	1m 55s	1m 21s	418MB	Y
`commit::Message`	41m 24s	3m 20s	1h 38m 3s	6GB	N

However, while the cache uses up disk space, reading an attribute from CBOR is potentially orders of magnitude faster than loading it from the store. On the other hand, when loading from the store is simple and the data is difficult to serialize (e.g. it consists of large string vectors) caching is not indicated. We have benchmarked and pre-tuned the database to cache only when it is clearly advantageous. Table 3 shows the performance impact of caching while extracting selected attributes on a dataset containing 130K projects and 44M commits. The table lists a few representative attributes in the first column. Columns two and three present what happens when the attribute is requested for the first time: how long it takes to extract it from the datastore and how long it takes to subsequently serialize it onto disk. The fourth and fifth columns show the impact of caching: how long it takes to read the argument from cache (e.g. when the query is re-executed or when another query requires the same attribute from the same datastore view) and how much disk space has to be devoted to the CBOR file. The final column shows our decision whether to cache this attribute or not.

3.3.4 Availability

While users can download their own datasets and run queries on them locally, doing so requires time and computational resources. Therefore, we also provide a procedure for running queries on our hardware using our incrementally updated dataset. A durable, publically available resource also fosters reproducibility.

The submission procedure plugs into the standard Rust toolkit. Queries are submitted as cargo crates. These crates include functions marked as individual queries via annotations which also specify the savepoint and subsets that the specific query expects. For convenience, we provide a template for query crates that works with the `cargo generate` command.[4] We also provide an accompanying `cargo djanco` command[5] which generates an execution harness around query functions. The harness is a small standalone Rust program that sets up the datastore and runs each query according to the specifications found in their annotations. The harness includes a commandline interface through which it can be executed with a specific dataset paths, output directory, and other parameters. We generate the harness for executing the query on our server, but it can be used to test queries locally as well.

As of this writing queries are scheduled manually by the authors. Users should contact us by email with a link to the repository. The query will undergo a manual inspection and will be executed on our hardware and dataset using the same generated harness as above. After the query is executed, a snapshot of the crate is created and stored it in the query archive. The snapshot contains the complete source code of all the queries, logs, the exact generated harness used for execution, and the results of all the queries – files generated to the designated output directory. Any result file exceeding 50MB is ignored (if a query produces large files we contact the user to advise on compaction or to negotiate different means of delivery).

In the future, we will extend our infrastructure to include a web API that will allow users to execute queries themselves. These queries will be expressed in a limited query language (to obviate security risks) and the volume of results will be limited. Queries and results will also be archived and accessible publicly with a receipt. Another extension we foresee is to extend the existing mechanism to allow automatic query execution. This would resemble our current process but it would remove the need for a manual check and emailing the authors

[4] `https://github.com/PRL-PRG/djanco-query-template#template`
[5] `https://github.com/PRL-PRG/cargo-djanco`

as submission could be automated. This option is contingent on our ability to create a static checker for incoming crates and sufficiently isolating them during execution.

Finally, storing user emails has privacy issues. we are considering whether it is appropriate to expose emails for external queries. If retaining emails becomes problematic, we may have to obfuscate the emails and replace them with numeric identifiers.

3.3.5 Reproducibility

To further support reproducibility, above and beyond the ability to deterministically run historical queries, every query executed by Djanco is stored in a public query archive. The query archive is a git repository hosted on GitHub.[6] Each query is hosted in a separate branch in the repository. We expect queries to undergo revisions. Each revision and execution results from that revision are archived as separate commits in a single branch. This produces a development history of the query.

Each query execution produces a *receipt* – a hash representing a specific commit in the archive repository representing the execution. The hash can be used to share queries (exactly as executed) and their results (exactly as produced). It can be used to retrieve the cargo crate and to re-execute the code (e.g. on a different dataset). Code re-execution is helped by the fact that queries are deterministic and the snapshot of the crate contains a list of all depedencies, a timestamp, a list of all subsets and all random seeds. The receipt for the queries in this paper is `da6ae7dd50565e84efbeac990f5788f383939014`.[7]

4 A Case Study: Of Bugs and Languages

The work's motivation is the claim that the *selection of inputs matters in empirical studies of software and that CodeDJ can assist researchers in that process.* We illustrate these points with a case study. We start from prior work, and show that input selection impacts scientific claims, and that CodeDJ allows rapid exploration of the input space.

The starting point is a Foundation of Software Engineering (FSE) paper published in 2014 [15].[8] One contribution of that work is to establish that some programming languages have a greater association with defects than others (RQ1 in [15]). Their methodology can be summarized as follows. For 17 popular languages, select 50 projects hosted on GitHub that have at least 28 commits. For each commit touching a file that contains code in one of the target languages, label the commit as bug-fixing if its message contains a bug-related keyword. Fit a Negative Binomial Regression (NBR) against the labeled data and obtain, for each language, a coefficient and a p-value. The coefficient indicates the strength of the association (positive means more bugs), and the p-value tells us about statistical significance (less than .05 means the coefficient is significant). The FSE paper concluded that TypeScript, Clojure, Haskell, Ruby and Scala were associated with *fewer* bugs, while C, C++, Objective-C, JavaScript, PHP and Python were associated with *more* bugs. The remaining languages did not have statistically significant coefficients.[9]

---

[6] `https://github.com/PRL-PRG/codedj-query-archive`

[7] `https://github.com/PRL-PRG/codedj-query-archive/tree/da6ae7dd50565e84efbeac990f5788f383939014`

[8] A revised version of the work appeared in the Communications of the ACM in 2017 with some issues fixed, notably the removal of TypeScript from the analyzed languages.

[9] These results were questioned, but the issues raised in [2] are orthogonal to the selection of inputs.

4.1 Corpus

For this experiment we created a datastore using stratified sampling of data available on GHTorrent. We started with 11,000 projects with at least 28 commits written in each of the 17 languages. For each language, we added 6,000 projects randomly selected from GitHub (including smaller projects). In total, our dataset had 172K projects with 28 or more commits and 230K projects in total. Only 3.8K large Erlang projects were available. The dataset has 47M unique commits (and 66M commits in total, suggesting a commit-duplication of 30%, high given forks were excluded). The datastore occupies 51GB on disk. Our goal was to have enough variety to represent the richness of GitHub. Unlike the FSE paper, which was written in 2013, our corpus goes all the way to 2020.

4.2 Random input selection

Our first experiment explores the distribution of possible analysis outcomes. For this, we repeatedly pick a random subset of 50 projects of each of the 17 languages and fit them with NBR. Fig. 4 shows the distribution of the coefficients obtained by 1000 such random selections compared to the results obtained in [15] (shown as a tick to the right of the distribution). Positive values indicate a higher association of the language with defects. The spread of each distribution is a measure of the sensitivity of the analysis to its inputs.

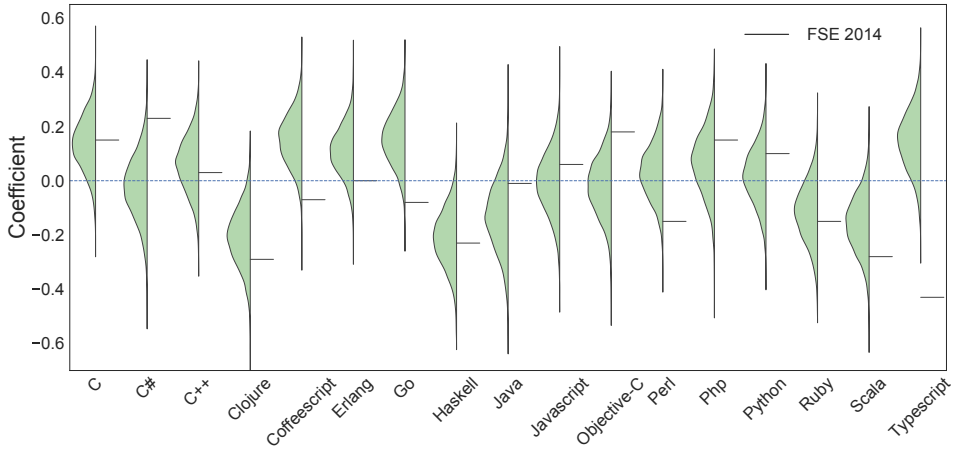


Figure 4 Random subsets.

Intuitively, consider the distribution of coefficients for Objective-C, it is roughly centered around 0. This means, that a random input is about equally likely to say that the language has a positive association with defects as a negative one. One could argue that picking close to the median of the distribution could give a representative answer. As we can see the FSE paper often picks subsets that are outliers; see the cases of CoffeeScript, Go, Perl, Scala and most strikingly TypeScript.

Discussion: As most distributions straddle the axis, random selection is likely to result in noisy conclusions. But, GitHub is noisy itself – for instance there is much code duplication, and the are many low quality projects. A random selection is not the appropriate choice for making conclusions about software developed by professionals. One could choose to mitigate selection bias by increasing the size of the sample; CodeDJ can be used to generate multiple random inputs, if the inputs agree, then our confidence in the results increases.

4.3 Observing change over time

As we have more data than was available in 2013, we can use CodeDJ to select inputs at various times. Here we create eight datasets, each containing data up to one of the years between 2013 and 2020. For simplicity, we only plot the distribution of coefficients for TypeScript. The original paper's coefficient was −.43 (shown as a red line). The graph clearly shows that the value was an outlier. The association with bugs shifted over time, increasing to a relatively stable position from 2016.

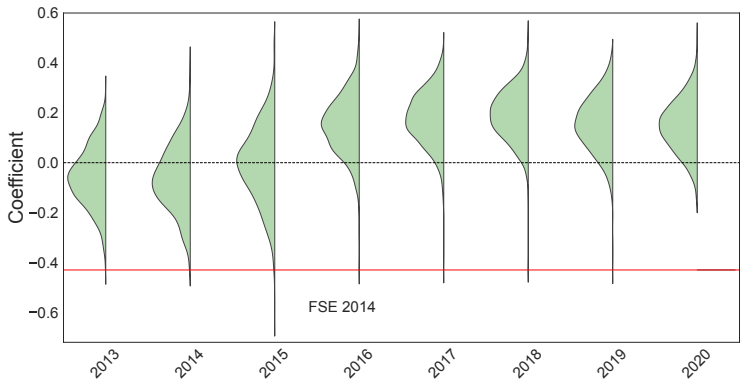


Figure 5 TypeScript over time.

While it is reasonable to expect variations from year to year, TypeScript experienced a rather large shift over a short period. The language was released in 2012, so there were few projects on GitHub in 2013. Furthermore, a number of human language translation files were misidentified as TypeScript; these files did not have bugs, biasing the result. The rising popularity of TypeScript quickly caused real code to crowd out the translation files, and the association with bugs settled to around 0.2.

Discussion: Using CodeDJ to prepare inputs at different time points can help researchers spot trends in the data. For some properties of interest one expects changes over time, for others changes may be an indication of bias that needs to be controlled for. For instance, one would expect the association with bugs of an established, popular, language to be stable.

4.4 Introducing domain knowledge

Choosing any subset of a larger population introduces bias, but this may be intentional, reflecting domain knowledge about the relative importance of observations. For instance, small projects with few commits may be less interesting as they correlate with student projects. These projects have fewer descriptive commit messages and their defects reflect beginner mistakes. It stands to reason to exclude such projects from consideration. Justifying the choice of any particular selection criterion is beyond the scope of our work. CodeDJ allows researchers to explore the impact of various subsets. Our next experiment looks at 6 different criteria for selecting projects and compares them to the original paper's criterion. The Djanco code for those queries is in Fig. 8 in the appendix.

- **Stars:** Pick projects with most stars. *Rationale:* starred projects are popular and thus likely to be well written and maintained. [**Used in FSE 2014**]
- **Touched Files:** compute #files changed by commits, pick projects that changed the most files. *Rationale:* indicative of projects where commits represent larger units of work.

- **Experienced Author:** experienced developers are those on GitHub for at least two years; pick a sample of projects with at least one experienced contributor. *Rationale:* less likely to be throw-away projects.
- **50% Experienced:** projects with two or more developers, half of which experienced. *Rationale:* focus on larger teams.
- **Message Size:** Compute size in bytes of commit messages; pick projects with the largest size. *Rationale:* empty or trivial commit messages indicate uninteresting projects.
- **Number of Commits:** Compute the number of commits; pick projects with the most commits. *Rationale:* larger projects are more mature.
- **Issues:** Pick projects with the most issues. *Rationale:* issues indicate a more structured development process.

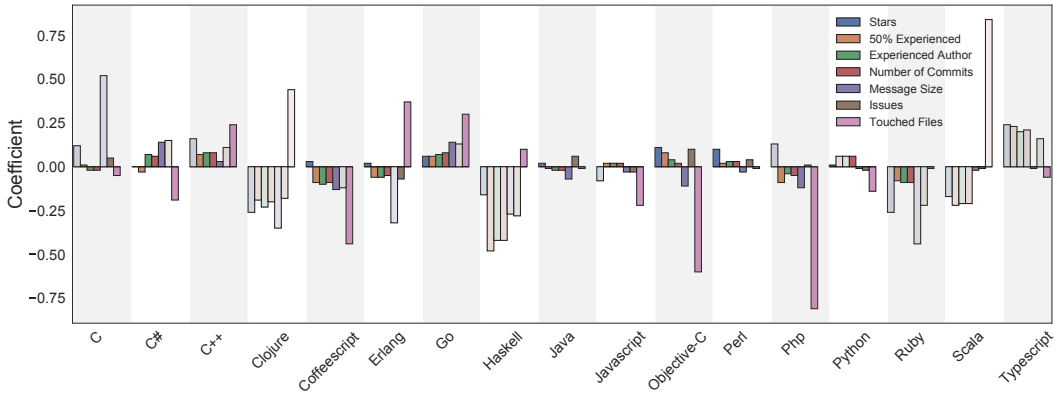


Figure 6 Domain knowledge.

Fig. 6 shows, for each language, the value of the coefficients (higher means more bugs); the queries returned 50 projects in each of the 17 target languages: Coefficients that are not statistically significant are shown in faded colors. If the input set did not matter for the model, one could expect the different queries to give roughly the same coefficients with the same significance. This is not the case. If we focus on how many languages have statistically significant coefficients: The touched files query is highly predictive, 14 of the languages are significant, but the coefficients are frequently opposite from those of other queries. Specifically, C is associated with slightly fewer bugs, so are C#, CoffeeScript, Java, JavaScript, Objective-C, Perl, PHP, Python, Ruby and TypeScript. On the other hand C++, Erlang, Go and Haskell are associated with more defects. This is striking as it goes against expectations. The stars query is the least informative. It only gives 7 statistically significant coefficients with remarkably low values.

Discussion: While some queries yield broadly similar conclusions, this is not the case for all. We stress the importance of understanding the selection criteria and its impact, as statistical significance should not be confused with validity. To help, CodeDJ provides distributions of various measures in the data, Fig. 7 visualizes the distribution of project sizes (left) and project age (right) for the entire dataset and for the various queries.

Looking at these distributions makes it clear that the queries return quite different projects. The experienced author and number of commits are remarkably similar and return projects that meet our expectations. The issues distribution is similar, which should raise red flags given that it frequently disagrees. The stars query returns many smaller projects. Finally, message sizes and touched files show distributions opposite to those expected. They favor

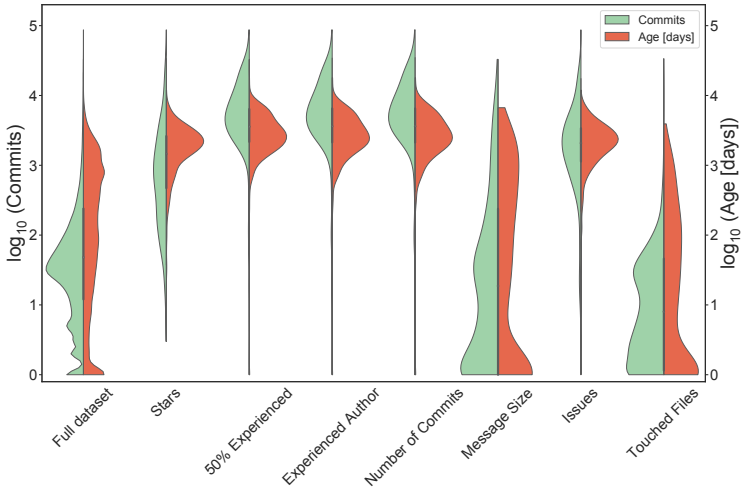


Figure 7 Project Size and Age Distributions.

degenerate young projects with few commits that are either verbose, or disproportionately large (touching over 100K files). This is reflected in the input sizes, ranging from 8M rows for the experienced author query to mere 79K rows of the touched files query. It is likely that these queries are "wrong" in the sense they do not return the population of interest. The figure also suggest that stars is a bad choice.

5 Conclusions

Finding projects on GitHub is akin to looking for the proverbial needle in a haystack. While having a wealth of data at our fingertips is an undeniable asset to empirical software engineering research, the sheer size of the code being hosted is a challenge to any data processing pipeline. Selecting manageable subsets of available projects can introduce subtle, but significant biases that, in turn, can influence or even invalidate the conclusion of the analysis being conducted. Our case study illustrates this problem – we demonstrate that by choosing various, apparently sensible, subsets of the data at hand, we can significantly change the observed association between programming languages and software defects.

This paper introduces CodeDJ, an infrastructure designed to support the reproducible specification of selection criteria for projects hosted on large-scale software repositories. Our implementation is geared towards GitHub. As GitHub is a living system undergoing constant change, ensuring reproducibility requires extra work. The same project downloaded today and last month may contain different code, different commit histories, or the project may disappear entirely. Our infrastructure mitigates this problem by building on a time-indexed, append-only datastore. Queries are expressed in a front-end database that can access a view of the data at a specific point in the history of the datastore.

For future work, three directions stand out: Expanding the datastore, improving the query evaluation performance, and extending accessibility of the our dataset. The dataset provided contains only a fraction of the data we expect to eventually need. As the data grows in volume, our downloading, storage, and processing capabilities will be put to the test and adjusted accordingly to ensure they scale up. We will explore how to ensure backwards compatibility and determinism of queries in the face of changes to the implementation, and to the data format (e.g. adding new information, such as issues, or new file kinds). In terms

of performance, our implementation does not try any optimizations of the query evaluation. We intend to parallelize queries and explore ideas from the database community regarding query compilation strategies. Finally, we plan on extending our infrastructure. We will create a web API and a limited query language to make our dataset more generally accessible. We will also investigate an infrastructure for automatic security checking and execution scheduling for query crates which would allow for their automated submission.

References

1 Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2019. doi:10.1145/3359591.3359735.

2 Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. On the impact of programming languages on code quality: A reproduction study. *ACM Trans. Program. Lang. Syst.*, 41(4):21:1–21:24, 2019. doi:10.1145/3340571.

3 T. F. Bissyande, F. Thung, D. Lo, L. Jiang, and L. Reveillere. Orion: A software project search engine with integrated diverse software artifacts. In *International Conference on Engineering of Complex Computer Systems*, 2013. doi:10.1109/ICECCS.2013.42.

4 Hudson Borges, André C. Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of GitHub repositories. *CoRR*, 2016. URL: http://arxiv.org/abs/1606.04984.

5 Andy Cockburn, Pierre Dragicevic, Lonni Besanc on, and Carl Gutwin. Threats of a replication crisis in empirical computer science. *Communications of the ACM*, 2020. doi:10.1145/3360311.

6 Roberto Di Cosmo and Stefano Zacchiroli. Software Heritage: Why and How to Preserve Software Source Code. *International Conference on Digital Preservation*, 2017. URL: https://hal.archives-ouvertes.fr/hal-01590958.

7 Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *International Conference on Software Engineering (ICSE)*, 2013. URL: http://dl.acm.org/citation.cfm?id=2486788.2486844.

8 Davide Falessi, Wyatt Smith, and Alexander Serebrenik. Stress: A semi-automated, fully replicable approach for project selection. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017. doi:10.1109/ESEM.2017.22.

9 Jesus M. Gonzalez-Barahona, Gregorio Robles, and Santiago Dueñas. Collecting data about FLOSS development: The FLOSSMetrics experience. In *International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS)*, 2010. doi:10.1145/1833272.1833278.

10 Georgios Gousios and Diomidis Spinellis. GHTorrent: GitHub's data from a firehose. In Michael W. Godfrey and Jim Whitehead, editors, *Working Conference on Mining Software Repositories (MSR)*, 2012. doi:10.1109/MSR.2012.6224294.

11 Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining GitHub. In *Working Conference on Mining Software Repositories (MSR)*, 2014. doi:10.1145/2597073.2597074.

12 P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. Adoption of software testing in open source projects–a preliminary study on 50,000 projects. In *European Conference on Software Maintenance and Reengineering*, 2013. doi:10.1109/CSMR.2013.48.

13 Crista Lopes, Petr Maj, Pedro Martins, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjà Vu: A map of code duplicates on GitHub. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017. doi:10.1145/3133908.

14 Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in software engineering research. In *Foundations of Software Engineering (FSE)*, 2013. doi:10.1145/2491411.2491415.

15 Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *International Symposium on Foundations of Software Engineering (FSE)*, 2014. doi:10.1145/2635868.2635922.

16 Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: scaling code clone detection to big-code. In *International Conference on Software Engineering (ICSE)*, 2016. doi:10.1145/2884781.2884877.

17 Gerald Schermann, Sali Zumberi, and Jürgen Cito. Structured information on state and evolution of dockerfiles on github. In *International Conference on Mining Software Repositories (MSR)*, 2018. doi:10.1145/3196398.3196456.

18 Christopher Vendome, Gabriele Bavota, Massimiliano Di Penta, Mario Linares-Vásquez, Daniel German, and Denys Poshyvanyk. License usage and changes: a large-scale study on GitHub. *Empirical Software Engineering*, 2016. doi:10.1007/s10664-016-9438-4.

19 Hadley Wickham, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain Franc ois, Garrett Grolemund, Alex Hayes, Lionel Henry, Jim Hester, Max Kuhn, Thomas Lin Pedersen, Evan Miller, Stephan Milton Bache, Kirill Müller, Jeroen Ooms, David Robinson, Dana Paige Seidel, Vitalie Spinu, Kohske Takahashi, Davis Vaughan, Claus Wilke, Kara Woo, and Hiroaki Yutani. Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686, 2019. doi:10.21105/joss.01686.

20 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010. doi:10.5555/1863103.1863113.

A Analysis with GitHub toolkits

Can users do without CodeDJ? Consider the case study queries: stars and touched files.

GitHub exposes a REST API that can return any object and its metadata. The API is limited. It allows filtering by language and sorting by stars, but not by touched files. Furthermore it only returns 1000 results. Therefore, we can't get directly the 17K projects of the case study. While repositories can be obtained by numeric IDs, given the rarity of some of languages such as Erlang means that a random sample would, in the worst case, end up sampling every project on GitHub.

Repository URLs can be retrieved with the `/repositories` query. Assuming 150M repositories, 1.5M queries are needed to find them all. The rate limit is 5K queries/user/hr, so this takes 12 days. We also need language and number of commits to perform stratified sampling. Getting languages is another 12 days. This can be done by getting a list of contributors and summing up their contributions. This only requires one query per repository, so another 12 days. Stratified sampling thus requires approximately a month.

The GitHub data is in JSON, which is not easy to query. One can convert it into a more useful format, such as a relational database. From there, one can retrieve top 50 most-starred projects in each language within that dataset with a query like:

```
select id from (
  select id, row_number() over(partition by language order by stars desc) as place
  from projects
) ranks
where place <= 50;
```

The second use case query requires ordering projects by average number of changes per commit. This requires information about all commits. The REST API can list commits, but not changes. To get those, the detailed metadata of each commit is need. This requires

one query per commit. With 66M commits, that is 550 days. Deduplicating commits before retrieval shaves this down to 391 days. Having retrieved the data, one can select projects:

```
select id from (
  select id, row_number() over(partition by lang order by avg_touched desc) as place
    from (
      select id, language as lang, avg(touched) as avg_touched
      from project_commits
      join (
        select commit_id, count(path_id) as touched
        from commit_changes
        group by commit_id
      ) touched on project_commits.commit_id = touched.commit_id
      join projects on projects.id = project_commits.project_id
      group by project_id, language
    ) projects
) ranks
where place <= 50;
```

The query is complex. An alternative is to update the data with precomputed attributes.

As the reader may have gathered using GitHub is impractical. An alternative is to use multiple sources of information. Project URLSs, stars and commit counts can be obtained from GHTorrent, commits can be obtained by cloning repositories and analyzing their logs locally. However, these sources have their own shortcomings. GHTorrent does not contain all information, and it can be out of date. For instance, we found commit and star counts off by orders of magnitude. Cloning repositories requires significant bandwidth. In addition, care must be taken with large projects as they can take weeks to analyze if approached naïvely. Gathering data never goes smoothly. The code will likely run for weeks even if massively parallel and then fail on some unexpected corner case. If one then continuously and incrementally update the obtained dataset, then one has essentially reinvented CodeDJ.

B Domain queries

Fig. 8 gives the queries used to inject domain knowledge in the analysis discussed in Sec. 4.

Stars:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .sort_by(project::Stars)
  .sample(Distinct(Top(50), MinRatio(project::Commits, 0.9)));
```

Touched Files:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .sort_by(Median(FromEach(project::Commits, Count(commit::Paths))))
  .sample(Distinct(Top(50), MinRatio(project::Commits, 0.9)));
```

Experienced Author:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .filter_by(AtLeast(Count(FromEachIf(project::Users,
                                      AtLeast(user::Experience,
                                              Duration::from_years(2)))), 1))
  .sort_by(Count(project::Commits))
  .sample(Distinct(Random(50, Seed(42)), MinRatio(project::Commits, 0.9)));
```

50% Experienced:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .filter_by(AtLeast(Count(project::Users), 2))
  .filter_by(AtLeast(Ratio(FromEachIf(project::Users,
                                      AtLeast(user::Experience,
                                              Duration::from_years(2))),
                           project::Users),
                     Fraction::new(1,2)))
  .sample(Distinct(Random(50, Seed(42)), MinRatio(project::Commits, 0.9)));
```

Message Size:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .sort_by(Mean(FromEach(project::Commits, commit::MessageLength)))
  .sample(Distinct(Top(50), MinRatio(project::Commits, 0.9)));
```

Number of Commits:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .sort_by(Count(project::Commits))
  .sample(Distinct(Top(50), MinRatio(project::Commits, 0.9)));
```

Figure 8 Domain queries.

Enabling Additional Parallelism in Asynchronous JavaScript Applications

Ellen Arteca
Northeastern University, Boston, MA, USA

Frank Tip
Northeastern University, Boston, MA, USA

Max Schäfer
GitHub, Oxford, UK

Abstract

JavaScript is a single-threaded programming language, so asynchronous programming is practiced out of necessity to ensure that applications remain responsive in the presence of user input or interactions with file systems and networks. However, many JavaScript applications execute in environments that do exhibit concurrency by, e.g., interacting with multiple or concurrent servers, or by using file systems managed by operating systems that support concurrent I/O. In this paper, we demonstrate that JavaScript programmers often schedule asynchronous I/O operations suboptimally, and that reordering such operations may yield significant performance benefits. Concretely, we define a static side-effect analysis that can be used to determine how asynchronous I/O operations can be refactored so that asynchronous I/O-related requests are made as early as possible, and so that the results of these requests are awaited as late as possible. While our static analysis is potentially unsound, we have not encountered any situations where it suggested reorderings that change program behavior. We evaluate the refactoring on 20 applications that perform file- or network-related I/O. For these applications, we observe average speedups ranging between 0.99% and 53.6% for the tests that execute refactored code (8.1% on average).

2012 ACM Subject Classification Software and its engineering → Automated static analysis; Software and its engineering → Concurrent programming structures; Software and its engineering → Software performance

Keywords and phrases asynchronous programming, refactoring, side-effect analysis, performance optimization, static analysis, JavaScript

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.7

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.5`

Funding E. Arteca and F. Tip were supported in part by the National Science Foundation grants CCF-1715153 and CCF-1907727. E. Arteca was also supported in part by the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

In JavaScript, asynchronous programming is practiced out of necessity: JavaScript is a single-threaded language and relying on asynchronously invoked functions/callbacks is the only way for applications to remain responsive in the presence of user input and file system or network-related I/O. Originally, JavaScript accommodated asynchrony using event-driven programming, by organizing the program as a collection of event handlers that are invoked from a main event loop when their associated event is emitted. However, event-driven programs suffer from event races [27] and other types of errors [21] and lack adequate support for error handling.

In response to these problems, the JavaScript community adopted promises [10, Section 25.6], which enable programmers to create chains of asynchronous computations with proper error handling. However, promises are burdened by a complex syntax where each element in a promise chain requires a call to a higher-order function. To reduce this burden, the `async`/`await` feature [10, Section 6.2.3.1] was introduced in the ECMAScript 8 version of JavaScript, as syntactic sugar for common usage patterns of promises. A function designated as `async` can await asynchronous computations (either calls to other `async` functions or promises), enabling asynchronous programming with minimal syntactic overhead.

The `async`/`await` feature has quickly become widely adopted, and many libraries have adopted promise-based APIs that enable the use of `async`/`await` in user code. However, many programmers are still unfamiliar with promises and `async`/`await` and are insufficiently aware of how careless use of these features may negatively impact performance. In particular, programmers often do not think carefully enough about when to create promises that are associated with initiating asynchronous I/O operations and when to `await` the resolution of those promises and trigger subsequent computations.

As JavaScript is single-threaded, it does not support multi-threading/concurrency at the language level. However, the placement of promise-creation operations and the awaiting of results of asynchronous operations can have significant performance implications because many JavaScript applications *execute in environments that do feature concurrency*. For example, a JavaScript application can interact with servers, file systems, or databases that can execute multiple operations concurrently. Therefore, in general, it is desirable to trigger asynchronous activities *as early as possible* and await their results *as late as possible*, so that a program can perform useful computations while asynchronous I/O requests are being processed in the environment.

In this paper, we use static interprocedural side-effect analysis [4] to detect situations where oversynchronization occurs in JavaScript applications. For a given statement s, our analysis computes sets $MOD(s)$ and $REF(s)$ of access paths [22] that represent sets of memory locations modified and referenced by s, respectively. We use this analysis to suggest how await-expressions of the form `await` e_{io} can be refactored, where e_{io} is an expression that creates a promise that is settled when an asynchronous I/O operation completes. Here, the idea is to "split" such await-expressions so that: (i) the promise creation is moved to the earliest possible location within the same scope and (ii) the awaiting of the result of the promise is moved to the latest possible location within the same scope. Like most static analyses for JavaScript, the side-effect analysis is unsound, so the programmer needs to ensure that program behavior is preserved, by reviewing the suggested refactorings carefully and running the application's tests.

We implemented the static analysis in CodeQL [2, 12], and incorporated it into a tool called *ReSynchronizer*[1] that automatically refactors I/O-related await-expressions. In an experimental evaluation, we applied *ReSynchronizer* to 20 open-source Node.js applications that perform asynchronous file-system I/O and asynchronous network I/O. Our findings indicate that, on these subject applications, our approach yields speedups ranging between 0.99% and 53.6% when running tests that execute refactored code (8.1% on average). We detected no situations where unsoundness in the static analysis resulted in broken tests.

In summary, the contributions of this paper are as follows:

- The design of a static side-effect analysis for determining MOD and REF sets of access paths, and the use of this analysis to suggest how I/O-related await-expressions can be refactored to improve performance,

[1] The source code of the tool and all of our data is available on GitHub

- Implementation of this analysis in a tool called *ReSynchronizer*, and
- An evaluation of *ReSynchronizer* on 20 open-source projects, demonstrating that our approach can produce significant speedups and scales to real-world applications.

The remainder of this paper is organized as follows. Section 2 reviews JavaScript's promises and `async`/`await` features. In Section 3, a real-world example is presented that illustrates how reordering await-expressions may yield performance benefits. Section 4 presents the side-effect analysis that serves as the foundation for our approach. Section 5 presents an evaluation of our approach on open-source JavaScript projects that use `async`/`await`. Related work is discussed in Section 6. Section 8 concludes and provides directions for future work.

2 Review of promises and `async/await`

This section presents a brief review of JavaScript's promises [10, Section 25.6] and the `async`/`await` feature [10, Section 6.2.3.1] for asynchronous programming. Readers already familiar with these concepts may skip this section.

A *promise* represents the result of an asynchronous computation, and is in one of three states. Upon creation, a promise is in the *pending* state, from where it may transition to the *fulfilled* state, if the asynchronous computation completes successfully, or to the *rejected* state, if an error occurs. A promise is *settled* if it is in the fulfilled or rejected state. The state of a promise can change only once, i.e., once a promise is settled, its state will never change again.

Promises are created by invoking the `Promise` constructor, which expects as an argument a function that itself expects two arguments, `resolve` and `reject`. Here, `resolve` and `reject` are functions for fulfilling or rejecting a promise with a given value, respectively. For example, the following code:

```
1 const p = new Promise(function(resolve, reject) {
2   setTimeout(function() { resolve(17); }, 1000);
3 });
```

creates a promise that is fulfilled with the value `17` after 1000 milliseconds.

Once a promise has been created, the `then` method can be used to register *reactions* on it, i.e., functions that are invoked asynchronously from the main event loop when the promise is fulfilled or rejected. Consider extending the previous example as follows:

```
4 p.then(function f(v) { console.log(v); return v+1; });
```

In this case, when the promise assigned to `p` is fulfilled, the value that it was fulfilled with will be passed as an argument to the resolve-reaction `f`, causing it to print the value `17` and return the value `18`.

The `then` function creates a promise, which is resolved with the value returned by the reaction. This enables the creation of a *promise chain* of asynchronous computations. For instance, extending the previous example with:

```
5 p.then(function(x) { return x+1; })
6  .then(function(y) { return y+2; })
7  .then(function(z) { console.log(z); })
```

results in the value `20` being printed.

The examples given so far only specify fulfill-reactions, but in general, care must be taken to handle failures. In particular, the promise implicitly created by calling `then` is rejected if an exception occurs during the execution of the reaction. To this end, the `catch` method can

be used to register reject-reactions that are to be executed when a promise is rejected. The `catch` method is commonly used at the end of a promise chain. For example:

```
8  p.then(function(x) { return x+1; })
9   .then(function(y) { throw new Error(); })
10  .then(function(z) { console.log(z); })
11  .catch(function(err) { console.log('error!'); })
```

results in `'error!'` being printed.

Recently, several popular libraries for performing I/O-related operations have adopted promise-based APIs. For example, `fs-extra` is a popular library that provides various file utilities, including a method `copy` for copying files. The `copy` function returns a promise that is fulfilled when the file-copy operation completes successfully, and that is rejected if an I/O error occurs, enabling programmers to write code such as:[2]

```
12 const fs = require('fs-extra')
13 fs.copy('/tmp/myfile', '/tmp/mynewfile')
14   .then(function() { console.log('success!'); })
15   .catch(function(err) { console.error(err); })
```

JavaScript's `async`/`await` feature builds on promises. A function can be designated as `async` to indicate that it performs an asynchronous computation. An `async` function f returns a promise: if f returns a value, then its associated promise is fulfilled with that value, and if an exception is thrown during execution of f, its associated promise is rejected with the thrown value. The `await` keyword may be used inside the body of `async` functions, to accommodate situations where the function relies on other asynchronous computations. Given an expression e that evaluates to a promise, the execution of an expression `await` e that occurs in the body of an async function f will cause execution of f to be suspended, and control flow will revert to the main event loop. Later, when the promise is fulfilled with a value v, execution of f will resume, and the await-expression will evaluate to v. In the case where the promise that e evaluates to is rejected with a value w, execution will resume and the evaluation of the await-expression will throw w as an exception that can be handled using the standard `try`/`catch` mechanism. Below, we show a variant of the previous example rewritten to use `async`/`await`.

```
16 async function copyFiles() {
17   try {
18     await fs.copy('/tmp/myfile', '/tmp/mynewfile')
19     console.log('success!')
20   } catch (err) {
21     console.error(err)
22   }
23 }
```

As is clear from this example, the use of `async`/`await` results in code that is more easily readable. Here, execution of `copyFiles` will be suspended when the await-expression on line 18 is encountered. Later, when the file-copy operation has completed, execution will resume. If the operation completes successfully, line 19 will execute and a message `'success!'` is printed. Otherwise, an exception is thrown, causing the handler on line 20 to execute.

As a final comment, we remark on the fact that it is straightforward to convert an existing event-based API into an equivalent promise-based API, by creating a promise that is settled when an event arrives. Various utility libraries exist for such "promisification" of event-driven APIs, e.g., `util.promisify` [14] and `universalify` [33].

---

[2] Example adapted from `https://www.npmjs.com/package/fs-extra`.

```
24 export async function getStatus(repository) {
25     const stdout = await gitMergeTree(repository)
26     const parsed = parsePorcelainStatus(stdout) (A)
27     const entries = parsed.filter(isStatusEntry) (B)
28
29     const hasMergeHead = await fs.pathExists(getMergeHead(repository))
30     const hasConflicts = entries.some(isConflict) (C)
31
32     const state = await getRebaseInternalState(repository)
33
34     const conflictDetails = await getConflictDetails(repository,
35                                   hasMergeHead, hasConflicts, state)
36
37     buildStatusMap(conflictDetails) (G)
38 }
```

(a)

```
39 async function getRebaseInternalState(repository) {
40     let targetBranch = await fs.readFile(getHeadName(repository))
41     if (targetBranch.startsWith('refs/heads/'))
42       targetBranch = targetBranch.substr(11).trim() (D)
43
44     let baseBranchTip = await fs.readFile(getOnto(repository))
45     baseBranchTip = baseBranchTip.trim() (E)
46
47     return { targetBranch, baseBranchTip } (F)
48 }
```

(b)

Figure 1 Example.

3 Motivating Example

We now present a motivating example that illustrates the performance benefits that may result from reordering await-expressions. The example was taken from Kactus[3], a git-based version control tool for design sketches. Figure 1(a) shows a function `getStatus` that is defined in the file `status.ts`[4]. As an `async` function, `getStatus` may depend on the values computed by other `async` functions, by awaiting such values in await-expressions. The code shown in Figure 1(a) contains four such await-expressions, on lines 25, 29, 32, and 34, which we now consider in some detail:

- The await-expression on line 25 invokes an async function `gitMergeTree` (omitted for brevity) that relies on the `dugite` and `child_process` libraries to execute a `git merge-tree` command in a separate process.
- The await-expression on line 29 calls an `async` function `pathExists` from the `fs-extra` package mentioned above, to check if a file `MERGE_HEAD` exists in the `.git` directory. `pathExists` is implemented in terms of the function `access` from the built-in `fs` package provided by the Node.js platform, which in turn triggers the execution of an OS-level file-read operation.
- The await-expression on line 32 calls an `async` function `getRebaseInternalState`, of which

[3] See `https://kactus.io/`.
[4] Some details not pertinent to the program transformation under consideration have been elided here. The complete source code can be found at `https://github.com/kactus-io/kactus`.

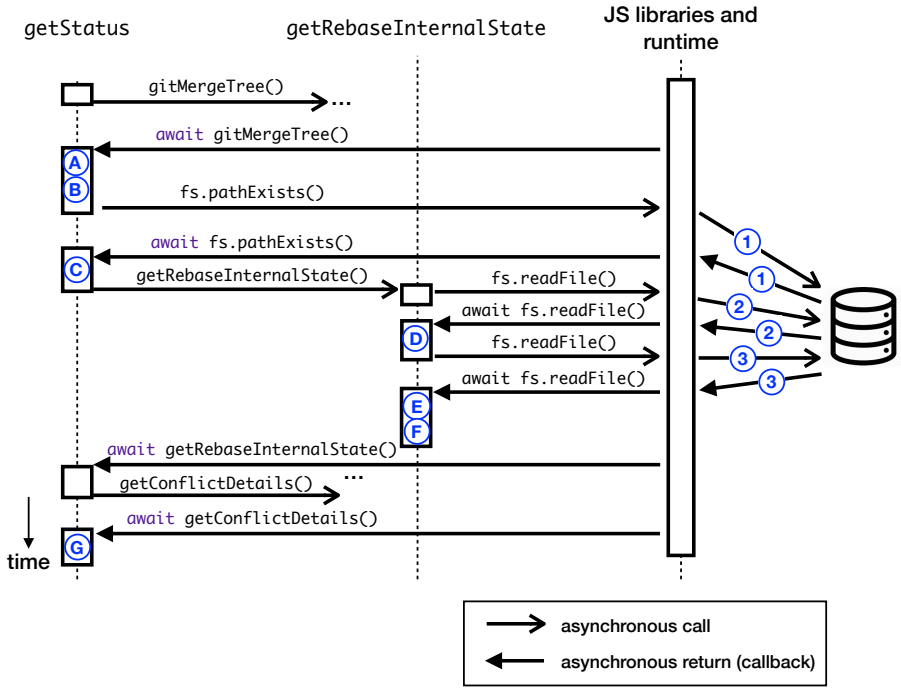


Figure 2 Visualization of the execution of `getStatus`.

we show some relevant fragments in Figure 1(b). Note in particular that two asynchronous file-read operations are performed on lines 40 and 44, using the `readFile` function from `fs-extra`. Each of these calls causes the execution of an OS-level file-read operation.

- The await-expression on line 34 invokes an `async` utility function `getConflictDetails` (omitted for brevity) to gather information about files that have merge conflicts.

Figure 2 shows a UML Sequence Diagram[5] that visualizes the flow of control during the execution of `getStatus`. In this diagram, labels (A) – (G) inside timelines indicate when code fragments labeled similarly in Figure 1 execute. Furthermore, labels (1) – (3) indicate when file I/O operations associated with the call to `fs.pathExists` on line 29 and with the two calls to `fs.readFile` in function `getRebaseInternalState` execute.

The leftmost timeline in the diagram depicts the execution of code fragments in the `getStatus` function itself. The middle timeline depicts the execution of function `getRebaseInternalState`. The timeline on the right, labeled "JS libraries and runtime" visualizes the execution of functions in JavaScript libraries such as `fs-extra` and other libraries that the application relies on such as `universalify` [33], `graceful-fs` [30], and libraries such as the `fs` file-system package that are included with the JS runtime.

Taking a closer look at the diagram, we can observe that the code fragments (A) and (B) will run before I/O operation (1) is initiated. Then, after I/O operation (1) has completed, code fragment (C) is evaluated. Next, when `getRebaseInternalState` is invoked, I/O operation (2) is initiated. After it has completed, code fragment (D) executes, which is followed in turn by I/O operation (3). When that operation completes, code fragments (E) and (F) execute,

[5] To prevent clutter, the diagram only shows asynchronous calls and returns and elides details that are not relevant to the example under consideration.

and finally code fragment (G) executes. Crucially, the use of `await` on lines 29, 32, 40, and 44 ensures that each file I/O operation must complete before execution can proceed. As a result, *the file I/O operations (1) – (3) execute in a strictly sequential order, where each operation must complete before the next one is dispatched.*

However, most JavaScript runtimes are capable of processing multiple asynchronous I/O requests concurrently. In this paper, we demonstrate that it is often possible to refactor JavaScript code in a way that enables for multiple I/O requests to be processed concurrently with the main program. The refactoring that we envision targets expressions of the form `await` e_{io}, where e_{io} is an expression that creates a promise that is settled when an asynchronous I/O operation completes. The expressions `await fs.pathExists(getMergeHead(repository))` on line 29 and `await getRebaseInternalState(repository)` on line 32 are examples of such expressions, as are the await-expressions on lines 40 and 44 in Figure 1(b).

Conceptually, the refactoring involves splitting an expression `await` e_{io} occurring in an async function f into two parts:

1. a local variable declaration `var` $t = e_{io}$ that starts the asynchronous I/O operation and that is placed *as early as possible* in the control-flow graph of f, and
2. an expression `await` t where the result of the asynchronous I/O operation is awaited and that is placed *as late as possible* in the control-flow graph of f.

We will make the notions "as early as possible" and "as late as possible" more precise in Section 4, but intuitively, the idea is that we want to move the expression e_{io} *before* any statement that precedes it – provided that this does not change the values computed or side-effects created at any program point. Likewise, we want to move the expression `await` t *after* any statement that follows it provided that this does not alter the values computed or side-effects created at any program point. Section 4 will present a static data flow analysis for determining when statements can be reordered.

Figure 3(a) shows how the `getStatus` function is refactored by our technique. As can be seen in the figure, the await-expression that occurred on line 29 in Figure 1(a) is split into the declaration of a variable `T1` on line 53 and an await-expression on line 60 in Figure 3(a). Likewise, the await-expression that occurred on line 32 in Figure 1(a) is split into the declaration of a variable `T2` on line 54 and an await-expression on line 59 in Figure 3(a).

The await-expression on line 25 cannot be split because it relies on `process.spawn` to execute a `git merge-tree` command in a separate process, and our analysis conservatively assumes that statements that spawn new processes have side-effects and thus cannot be reordered (this is discussed in detail in Section 4.4). Furthermore, the await-expression on line 34 was not reordered because it references the variable `state` defined on the previous line, and it defines a variable `conflictDetails` that is referenced in the subsequent statement, so any reordering might cause different values to be computed at those program points.

The two await-expressions in Figure 1(b) can also be split, and the resulting refactored code is shown in Figure 3(b).

Figure 4 shows a UML Sequence diagram that visualizes the execution of the refactored `getStatus` method. As can be seen in the figure, the I/O operation labeled (1) is now initiated after code fragment (A) has been executed but before code fragment (B) executes. However, since the result of this I/O operation is not needed until after code fragment (C) has executed, this I/O operation can now execute *concurrently* with I/O operations (2) and (3). Additional potential for concurrency is enabled by starting I/O operation (3) before awaiting the result of I/O operation (2). Note that, as a result of splitting await-expressions and reordering statements, the labeled code fragments now execute in a slightly different order: (A), (D), (E), (F), (B), (C), (G). Our static analysis, defined in Section 4 inspects the MOD and REF sets of

```
49 export async function getStatus(repository) {
50     const stdout = await gitMergeTree(repository)
51     const parsed = parsePorcelainStatus(stdout) (A)
52
53     let T1 = fs.pathExists(getMergeHead(repository))
54     let T2 = getRebaseInternalState(repository)
55
56     const entries = parsed.filter(isStatusEntry) (B)
57     const hasConflicts = entries.some(isConflict) (C)
58
59     const state = await T2
60     const hasMergeHead = await T1
61     const conflictDetails = await getConflictDetails(repository,
62                                     hasMergeHead, hasConflicts, state)
63
64     buildStatusMap(conflictDetails) (G)
65 }
```

(a)

```
66 async function getRebaseInternalState(repository) {
67     let T3 = fs.readFile(getHeadName(repository))
68     let T4 = fs.readFile(getOnto(repository))
69     let targetBranch = await T3
70     if (targetBranch.startsWith('refs/heads/'))
71       targetBranch = targetBranch.substr(11).trim() (D)
72
73     let baseBranchTip = await T4
74     baseBranchTip = baseBranchTip.trim() (E)
75
76     return { targetBranch, baseBranchTip } (F)
77 }
```

(b)

Figure 3 Example, reordered.

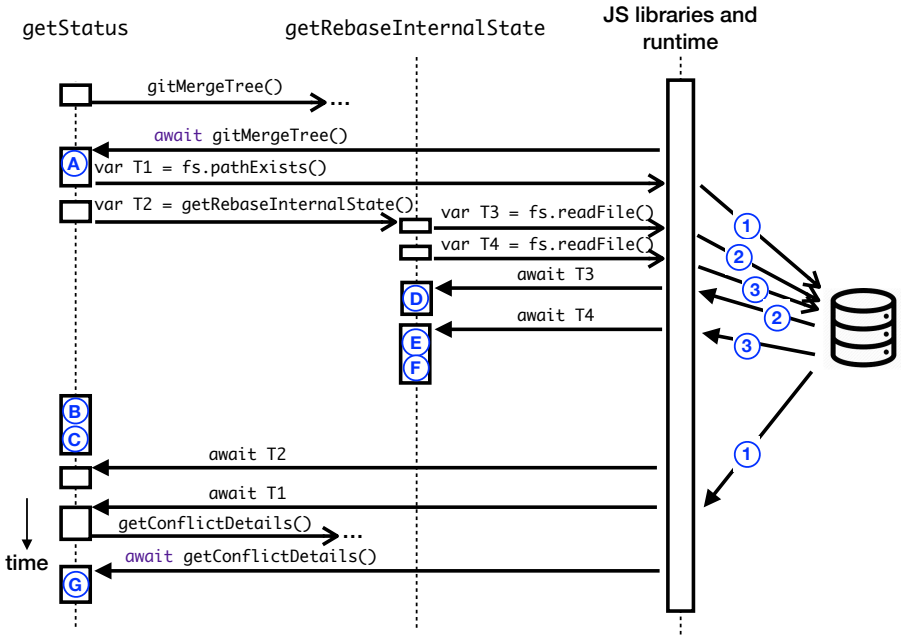


Figure 4 Visualization of the execution of `getStatus` after reordering.

memory locations modified and referenced by statements to determine when reordering is safe. The analysis is unsound, and may potentially suggest reorderings that change program behavior, so programmers need to review the suggested changes carefully and run their tests to ensure that behavior is preserved. In practice, however, we have not encountered any cases where invalid reorderings were suggested, as we will discuss in Section 5.3 .

At this point, the reader may wonder whether the additional concurrency enabled by the suggested transformation results in performance improvements. For the Kactus project from which the example was taken, a total of 72 I/O-related await-expressions were reordered by our technique, including the ones discussed above. Of the 799 tests associated with Kactus, 172 execute at least one reordered await-expression. For these impacted tests, we observed an average speedup of 7.2%. We discuss our experimental results in detail, in Section 5.

4 Approach

This section presents a static analysis for determining how await-expressions can be reordered to reduce over-synchronization. The analysis determines whether reordering adjacent statements may impact program behavior by determining the side-effects of each statement. Here, the *side-effects* of statements are defined in terms of MOD and REF sets [4] of access paths [22]. Below, we will define these concepts before introducing predicates that specify when statements can be reordered.

4.1 Access paths

An *access path* represents a set of memory locations referred to by an expression in a program. The access path representation that we use is based on the work by Mezzetti et al. [22]: starting from a root, an access path records a sequence of property reads, method calls and function parameters that need to be traversed to arrive at the designated locations. It is often also useful to view access paths as representing a set of values, namely those values that are stored in these locations at runtime. Access paths a conform to the following grammar:

a	::=	**root**	a root of an access path
	|	$a.f$	a property f of an object represented by a
	|	$a()$	values returned from a function represented by a
	|	$a(i)$	the i^{th} parameter of a function represented by a
	|	$a_{\textbf{new}}()$	instances of a class represented by a

Mezzetti et al. developed access paths to abstractly represent objects originating from a particular API. As such, their **root** was always of the form **require**(m)[6]. We additionally allow variables as roots, including both global variables and local variables, with the latter also covering function parameters including the implicit receiver parameter `this`.

▶ **Example 4.1.** We give a few examples of access paths:

- The local variable `targetBranch` declared on line 40 in Figure 1 is represented by the access path `targetBranch`.
- The argument `'refs/heads/'` in the method call `targetBranch.startsWith('refs/heads/')` on line 41 is represented by the access path `targetBranch.startsWith(1)`.

[6] This represents an import of package m. For simplicity, we use this same notation to represent packages imported using `require` or `import`.

- The property-access expression `fs.pathExists` on line 29 is represented by the access path `require(fs-extra).pathExists`.

Note that access paths are not canonical: due to aliasing, it is possible for multiple access paths to represent the same memory locations. This may give rise to unsoundness in the analysis, as will be discussed in Section 4.10.

4.2 MOD and REF

Intuitively, for a given statement or expression s, $MOD(s)$ is a set of access paths representing locations modified by s and $REF(s)$ is a set of access paths representing locations referenced by s. If s is a compound statement or expression such as a block, if-statement, or while-statement, $MOD(s)$ and $REF(s)$ include all access paths modified/referenced in any component of s, respectively. Furthermore, if s includes a function call $e.f(\cdots)$, $MOD(s)$ and $REF(s)$ include all access paths modified/referenced in any statement in any function transitively invoked from this call site[7].

When a statement s contains an assignment to an access path a, the set $MOD(s)$ contains a and all access paths that are rooted in a. However, note that we limit the set of access paths in $MOD(s)$ to those that are explicitly referenced in the program. To understand why this must be the case, consider a scenario where a is a variable containing a string. Such a variable has all properties that are defined on strings[8]. As one particular example, consider the `toString` function defined on strings. Since $a.toString()$ is rooted in a, $MOD(s)$ should include $a.toString()$. The result of $a.toString()$ is also a string, which means that $a.toString().toString()$ is another valid access path rooted in a, and should be included in $MOD(s)$. This could be repeated ad infinitum, and is only one possible example of such an infinite recursive process. So, to ensure that $MOD(s)$ and $REF(s)$ are always finite sets, they only include access paths that actually occur in the program.

Note that, in JavaScript, it is also possible to access properties dynamically, with expressions of the form $e[p]$, where p is a value computed at run time. In such cases, our analysis cannot statically determine which of e's properties is specified by p, and so we conservatively assume that *all* properties of e are accessed (i.e., all access paths rooted in e).

▶ **Example 4.2.** Consider the assignment statement on line 40 in Figure 1.

```
let targetBranch = await fs.readFile(getHeadName(repository))
```

Since we are assigning to `targetBranch`, this statement modifies `targetBranch` *and* all access paths rooted in `targetBranch`. From a quick glance at the code, we can see that two properties of `targetBranch` are accessed (`startsWith` and `substr`) and called as methods, and the `trim` method is called on the result of calling `substr` (and none of these has any further properties accessed). The assignment also contains a call to `getHeadName` – the function body is elided for brevity, but suffice it to say that `getHeadName` does not modify its `repository` argument or any global variables. Taking these considerations into account, the following MOD set is computed for the statement on line 40:

{ `targetBranch`, `targetBranch.startsWith`, `targetBranch.startsWith()`, `targetBranch.substr`, `targetBranch.substr()`, `targetBranch.substr().trim`, `targetBranch.substr().trim()` }

[7] Note that for brevity, when describing modification/reference of the locations abstractly represented by an access path, we refer to it as modification/reference of the access path itself.

[8] See `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String`.

The REF set includes all access paths referenced in the assignment, which includes the call to `fs.readFile` that is represented by the access path **require**(`fs-extra`).`readFile`(), the function `getHeadName`, and the variable `repository`. In the implementation of function `getHeadName`, there is a call to `fs.pathExists`, another to `Path.join`, and an access to the `path` property of the `repository` object. Therefore, the REF set for the statement is:

{ **require**(`fs-extra`), **require**(`Path`), **require**(`fs-extra`).`readFile`, **require**(`fs-extra`).`readFile`(), **require**(`fs-extra`).`pathExists`, **require**(`fs-extra`).`pathExists`(), **require**(`Path`).`join`, **require**(`Path`).`join`(), `repository`, `repository.path` }

Note that, for a given statement s, $MOD(s)$ and $REF(s)$ do not include access paths rooted in local variables, parameters or `this` parameters in scopes disjoint from the scope of s. For example, for the statement on line 32 where we see a call to `getRebaseInternalState`, the MOD set does not include an access path `targetBranch` for the local variable `targetBranch` modified in that function because it has no effect on the calling statement.

4.3 Determining whether statements are independent

In order to determine whether two adjacent statements s_1 and s_2 can be reordered, we need to determine whether doing so might change the values computed at either statement. We consider statements s_1 and s_2 *data-independent* if all of the following criteria are satisfied:

1. $MOD(s_1) \cap MOD(s_2) = \emptyset$
2. $MOD(s_1) \cap REF(s_2) = \emptyset$
3. $REF(s_1) \cap MOD(s_2) = \emptyset$

If s_1 and s_2 are not data-independent, then we will say that they are *data-conflicting*.

▶ **Example 4.3.** We discussed the MOD set for the statement at line 40 in Figure 1 in Example 4.2. Similarly, the statement on line 44 is an assignment to variable `baseBranchTip`, whose MOD set consists of {`baseBranchTip`, `baseBranchTip.trim`, `baseBranchTip.trim()`}. Since neither of these statements is modifying data that the other is modifying or referencing, these statements are *data-independent*. Note that they do have an overlap in the REF sets: both statements include calls to `fs.readFile`, and access the variable `repository`. However, since these accesses are read-only, the order in which they execute does not need to be preserved. Indeed, in Figure 3, we see that, in the reordered code, the `await` for the `targetBranch` assignment is moved *after* the `baseBranchTip` assignment.

Since the statement on line 44 has `baseBranchTip` in its MOD set, it *data-conflicts* with the statement on line 45 which uses the value of variable `baseBranchTip`, indicating that these statements cannot be reordered. Indeed, in Figure 3, we see that the `await` for the assignment of `baseBranchTip` remains *before* the reference to `baseBranchTip` on line 74.

Note that, since access paths are not canonical, data independence is not, strictly speaking, a sound criterion for reorderability: if two statements modify the same location under different access paths, we will consider them to be data independent, but reordering them may be unsafe. This issue and other factors that may impact soundness are discussed in Section 4.10.

4.4 Environmental side effects

So far, we have only considered side-effects consisting of referencing and modifying locations through variables and object properties. However, statements may also have side-effects beyond the state of the program itself, such as modifications to file systems, or the environment in which the program is being executed. Our approach to handling such side-effects is to

Table 1 Functions with environment-specific MOD side-effects.

Environment	Function names
`__FILE_SYSTEM__`	`fs.write*` (i.e. `fs.write`, `fs.writeSync`, `writeFile`, etc)
`__FILE_SYSTEM__`	`fs.append*` (i.e. `fs.append`, `appendFile`, etc)
`__FILE_SYSTEM__`	`fs.unlink`, `fs.remove`, `fs.rename`, `fs.move`, or `fs.copy`
`__FILE_SYSTEM__`	`fs.mkdir` or `fs.rmdir` or `fs.rimraf`
`__FILE_SYSTEM__`	`fs.output*` (i.e. `fs.output`, `fs.outputFileSync`, etc)
`__FILE_SYSTEM__`	`process.chdir`
`__NETWORK__`	`network.start` or `network.stop` or `network.launch`
`__NETWORK__`	`network.write`, or `network.load` (a write to the contents of a page)
`__NETWORK__`	`network.goto` (for changing pages in `puppeteer`; it is analagous to `chdir` for `fs`)

model them in terms of MOD and REF sets for (pseudo-)variables. We distinguish two types of special side effects: *global* and *environment-specific*, which we discuss below.

Global environmental side-effects

We say that a statement s has a *global side-effect* if it could affect any of the data in the program or its environment. In such cases, our analysis infers that $MOD(s) = \top$ and $REF(s) = \top$, where $\top$ is the set containing all access paths computed for the program. Currently, our analysis flags the following functions as having global side-effects: `eval`, `exec`, `spawn`, `fork`, `run`, and `setTimeout`. All but the last of these functions may execute arbitrary code and `setTimeout` is often used to explicitly force a specific execution order[9].

Environment-specific side-effects

We say a statement has an *environment-specific side-effect* if it can affect a specific aspect of the program's run-time environment, such as the file system or network. Environment-specific side-effects are modeled in terms of MOD and REF sets for pseudo-variables that are introduced for the aspect of the environment under consideration.

The experiments reported on in this paper focus on applications that access the file system or a network and we model these environments using pseudo-variables `__FILE_SYSTEM__` and `__NETWORK__` respectively.

Our current implementation flags a statement as having an environment-specific MOD side-effect if it consists of a call to any of the functions listed in Table 1. For each of these operations, the MOD sets will include the corresponding environment pseudo-variable. For example, the first row reads as follows: a statement including any function starting with `write` (i.e. `write`, `writeSync`, `writeFile`, etc.) that originates from a file system-dependent package will include the pseudo-variable `__FILE_SYSTEM__` in its MOD set.

Any other operations that reference the environments will have their REF set include the corresponding pseudo-variable (e.g., `fs.readFile` references `__FILE_SYSTEM__`, and `express.get` references `__NETWORK__`)[10]. As a result, no statements that reference an environment can be reordered around a call that may modify that environment. For example, no file read will ever be reordered around a file write, since the file read statements have `__FILE_SYSTEM__` in

[9] While conducting our experiments, we ran into cases where reordering awaits around a call to `setTimeout` caused changes in program behavior because the execution order was modified.

[10] This full list is included in a table analogous to Table 1 in the supplementary materials.

the REF set and the file write statements have `__FILE_SYSTEM__` in the MOD set[11]. However, any two file reads can be reordered (as seen in our motivating example), since there will never be a data conflict between read-only operations.

Algorithm 1 Predicate for determining if an access path a is modified by a statement s.

Input: s statement and a access path
Result: `True` if s modifies a, `False` otherwise

predicate $MOD(s, a)$
  // (i) base case: direct modification of a
  (s has environmental side-effect a $\vee$ s declares or assigns to a)
  $\vee$ // recursive cases...
    // (ii) check if there's a statement nested in s (in the AST) that modifies a
    $\exists\, s_{in},\ nestedIn(s_{in}, s) \wedge MOD(s_{in}, a)$
    // (iii) check if s modifies a base path of a
    $\vee\, \exists\, b,\ b.p == a \wedge MOD(s, b)$
    // (iv) check if s modifies a property of a using a dynamic property expression
    $\vee$ s assigns to $a[p]$
    // (v) check if s contains a call to a function that modifies a
    $\vee\, \exists\, f,\ calledIn(f, s) \wedge \exists\, s_f \in f_{body},$
      // direct modification of a in the function
      $MOD(s_f, a)$
      $\vee$ // parameter alias to a is modified in the function
      a is f's i^{th} argument $\wedge$ $\exists\, a_{pi},\ MOD(s_f, a_{pi}) \wedge a_{pi}$ is f's i^{th} parameter
end predicate

4.5 Computing MOD and REF sets

Algorithm 1 shows our algorithm for computing MOD sets[12], expressed as a predicate *MOD*. The *MOD* predicate states that statement s modifies access path a if one of the following conditions holds: (i) s modifies a directly in an assignment or in the initializer associated with a declaration, or via an environment-specific side effect, (ii) there is a statement nested inside s that modifies a, (iii) s modifies a base path of a (i.e., $a == b.p$, and s modifies b), (iv) s modifies a property of a using a dynamic property expression p, or (v) s consists of a call to a function f, the body of f contains a statement s_f, and either s_f modifies a or s_f modifies a parameter of f that is bound to a.

Algorithm 2 Predicate for determining if two statements have overlapping MOD/REF sets.

Input: s_1 and s_2 statements
Result: boolean indicating if s_1 and s_2 are data-independent

predicate $dataIndependent(s_1, s_2)$
  $\forall a,\ MOD(s_1, a) \implies \neg MOD(s_2, a)$
  $\wedge\ \forall a,\ MOD(s_1, a) \implies \neg REF(s_2, a)$
  $\wedge\ \forall a,\ REF(s_1, a) \implies \neg MOD(s_2, a)$
end predicate

[11] We have taken this conservative approach because, in many cases, it is not possible to determine precisely which files are being accessed because names of accessed files are specified with string values that may be computed at run time.
[12] REF sets are computed analogously; pseudocode of the REF algorithm is in the supplementary material.

Algorithm 3 Predicate for determining if two statements can be swapped.

Input: s_1 and s_2 statements
Result: boolean indicating if the statements can be exchanged

predicate $exchangeable(s_1, s_2)$
  $dataIndependent(s_1, s_2)$
  $\wedge\ \neg isControlFlowStmt(s_1) \wedge \neg isControlFlowStmt(s_2)$
  $\wedge\ inSameBlock(s_1, s_2)$
end predicate

4.6 Determining whether statements can be exchanged

As a first step towards determining reordering opportunities, Algorithm 2 defines a predicate for determining if two statements are data-independent, by checking that they do not have conflicting side-effects. This predicate operationalizes the condition that was specified in Section 4.3. However, data-independence is by itself not a sufficient condition for statements being exchangeable. Algorithm 3 shows a predicate *exchangeable* that checks if two statements s_1 and s_2 are exchangeable by checking that: (i) they are data independent, (ii) neither is a control-flow construct such as `return` or the test condition of an `if` or loop, and (iii) they occur in the same block. Condition (iii) expresses that we do not move statements into a different scope, to avoid problems that might arise due to name collisions. As part of future work, we plan to incorporate strategies from existing refactorings [28] to relax this condition so that statements can be moved into different scopes.

Algorithm 4 Predicate for determining if statement s can be reordered above another statement s_{up}.

Input: s and s_{up} statements
Result: boolean indicating if s can be reordered above s_{up}

predicate $stmtCanSwapUpTo(s, s_{up})$
  $s == s_{up}$ `// base case`
  $\vee$ `// recursive case`
  $\exists\ s_{mid},\ (\ stmtCanSwapUpTo(s, s_{mid}) \wedge$
    $s_{up}.nextStmt == s_{mid} \wedge$
    $exchangeable(s, s_{up})\ )$
end predicate

Algorithm 5 Predicate for finding the earliest statement above which s can be placed.

Input: s and **result** statements
Result: boolean indicating if **result** is the earliest statement above which s can be swapped

predicate $earliestStmtToSwapWith(s,$ **result**$)$
  `// find the earliest statement s can swap above (min by source code location)`
  result $== min($ all stmts s_i where $inSameBlock(s, s_i) \wedge stmtCanSwapUpTo(s, s_i))$
end predicate

4.7 Identifying reordering opportunities

We are now in a position to present our algorithm for identifying reordering opportunities. The analysis for determining earliest point above which a statement can be placed is symmetric to

that for the latest point below which a statement can be placed, so without loss of generality we will focus on the case of determining the earliest point. Our solution for this problem takes the form of two predicates, *stmtCanSwapUpTo* and *earliestStmtToSwapWith* [13].

Algorithm 4 defines a predicate *stmtCanSwapUpTo* that associates a statement s with an earlier statement s_{up} above which it can be reordered. This predicate relies on the predicate *exchangeable* to determine if it can be swapped with each statement in between s and s_{up}. If one of these intermediate statements data-conflicts with s then reordering is not possible.

The predicate *earliestStmtToSwapWith* defined in Algorithm 5 uses *stmtCanSwapUpTo* to find the earliest statement above which a statement can be placed.

We apply this predicate to statements containing I/O-dependent await-expressions, to identify reordering opportunities that can enable concurrent I/O. Here, an await-expression is considered *I/O-dependent* if it (transitively) invokes functions originating from one of the (many) npm packages that make use of the file system or work across a network. I/O dependency is determined by analyzing the call graph, much like how we compute MOD and REF sets. In particular, for statement s we look for calls to I/O-related package functions explicitly in s, or in a function transitively called by s. In terms of access paths, these calls correspond to function call access paths rooted in a **require**(m) for some I/O-dependent package m. This algorithm is included in pseudocode in the supplementary materials.

4.8 Program transformation

As discussed in Section 3, the execution of an await-expression `await` e_{io} involves two key steps: the *creation of a promise*, and *awaiting its resolution*. The creation of the promise kicks off an asynchronous computation, and our goal is to move it as *early* as possible, so as to maximize the amount of time where it can run concurrently with the main program or other concurrent I/O. On the other hand, we want to await the resolution of the promise *as late as possible*, for the same reason. We achieve this objective by splitting the original await-expression into two statements `var t =` e_{io} and `await t`, and using our analysis to move the former as early as possible, and the latter as late as possible. The example given previously in Section 3 illustrates an application of this refactoring to a real code base.

4.9 Implementation

We implemented our approach in a tool named *ReSynchronizer*[14]. The static analysis algorithm, as presented in Section 4, is implemented using approximately 1,600 lines of QL [2], building on extensive libraries for writing static analyzers provided by CodeQL [13]. In particular, we rely on existing frameworks for dataflow analysis and call graphs, and on an implementation of access paths that we extended to suit our analysis, as discussed. Note that the CodeQL standard library caps access paths at a maximum length of 10; this could lead to MOD/REF for very long paths not being accounted for, which is a source of potential unsoundness (see Section 4.10). The CodeQL representation of local variables also relies on single static assignment (SSA), enabling us to regain some precision that would be lost in a purely flow-insensitive analysis.

Once *ReSynchronizer* has determined the await-expressions that are to be reordered and where they should be moved to, the next stage of the tool is to create the transformed program so that the programmer can review the changes and run the tests. The actual

[13] Pseudocode for *stmtCanDownUpTo* and *latestStmtToSwapWith* included in the supplementary material.
[14] *ReSynchronizer* will be made available as an artifact.

reordering is done by splitting and moving nodes around in a parse tree representation of the program. We implemented this in Python, and use the pandas library[25] to store our list of statements to reorder in a dataframe over which we can efficiently apply transformations.

4.10 Soundness of the Analysis

As mentioned, it is possible for multiple access paths to represent the same memory locations because our analysis only accounts for aliasing resulting from passing an argument to a function (i.e., where an argument is referenced by the parameter name in the function's scope). As a result, our analysis may deem two statements to be data-independent when they are accessing the same memory locations, which may result in invalid orderings being suggested. Unsoundness may also arise because the underlying CodeQL infrastructure limits the lengths of access paths to a maximum length of 10, and because of unsoundness in the call graph that is used to compute MOD and REF sets. For example, the use of dynamic features such as `eval` may give rise to missing edges in the call graph, causing the absence of access paths in the MOD and REF sets, which in turn may result in invalid reordering suggestions. Section 5.3 reports on how often unsoundness has been observed in practice in our experimental evaluation.

5 Evaluation

In this section, we apply our technique to a collection of open-source JavaScript applications to answer the following research questions:

RQ1 (Applicability). How many await-expressions are identified as candidates for reordering?

RQ2 (Soundness). How often does *ReSynchronizer* produce reordering suggestions that are not behavior-preserving?

RQ3 (Performance Impact). What is the impact of reordering await-expressions on runtime performance?

RQ4 (Analysis Time). How much time does *ReSynchronizer* take to analyze applications?

5.1 Experimental Methodology

To answer the above research questions, we applied *ReSynchronizer* to 20 open-source JavaScript applications that are available from GitHub. We analyzed these applications, applied the suggested refactorings, and measured the performance impact of the refactoring by comparing the running times of the application's tests before and after the refactoring.

Selecting subject applications

To be a suitable candidate for our technique, an application needs to apply the `async`/`await` feature to promises that are associated with I/O. Furthermore, to conduct performance measurements, we need to be able to observe executions in which the reordered await-expressions are evaluated. To this end, we focus on applications that have a test suite that we can execute, and monitor test coverage to observe whether await-expressions are executed.

To identify projects that satisfy these requirements, we wrote a CodeQL query that identifies projects that contain await-expressions in files that import a file system I/O-related

Table 2 Summary of GitHub projects we're using for experiments.

Project	LOC	#fun (async)	#await (IO)	#test	IO	Brief description
kactus	134k	12321 (335)	2430 (1201)	799	FS	Version control for sketch
webdriverio	19k	1393 (81)	1815 (126)	1884	FS	Node WebDriver automated testing
desktop	145k	12926 (284)	2450 (1232)	837	FS	Github desktop app
fiddle	6.4k	346 (37)	479 (108)	609	FS	Tool for small Electron experiments
nodemonorepo	4.3k	310 (31)	214 (160)	499	FS	Management of nodejs env/packages
zapier-...	5.6k	320 (26)	136 (59)	36	FS	CLI tool for zapier applications
wire-desktop	5.9k	294 (41)	553 (236)	37	FS	Desktop app for wire messenger
cspell	9.8k	676 (70)	367 (226)	954	FS	Spell checker for code
sourcecred	32k	2424 (186)	840 (191)	1824	FS	Reputation networks for OSS
bit	50k	5738 (251)	2488 (2144)	405	FS	Component collaboration platform
vscode-psl	8.7k	681 (87)	665 (406)	450	FS	Profile Scripting Lang VSCode plugin
gatsby	81k	3047 (598)	4145 (821)	2708	FS	Web framework built on React
jamserve	33k	5141 (4019)	10825 (1067)	3883	FS	Audio library server
get	404	29 (6)	40 (29)	50	FS	Download Electron release artifacts
cucumber-js	11k	655 (115)	532 (31)	445	FS	Cucumber for JS
sapper	7.9k	675 (17)	155 (43)	151	NW	Web app framework on svelte
svelte	56k	3652 (15)	151 (18)	3165	NW	Declarative webapp construction
reflect	124	18 (7)	19 (6)	16	NW	Reflect directory contents
m...-redux	76k	6664 (560)	1962 (719)	1331	NW	Redux for mattermost
enquirer	5.8k	526 (54)	395 (15)	175	NW	Stylish CLI prompts

package[15] or a network I/O-related package[16], and ran it over all 85k JavaScript projects available on GitHub's `LGTM.com` site. This resulted in a list of 42,378 candidate projects. To further narrow the list, we filtered for projects that contain at least 50 await-expressions in files that import a file system or network I/O-related package. This left us with 1,200 candidate projects.

From these candidates, we then randomly selected a project, cloned its repository, and attempted to build the project by running the setup code. If the build was successful, we ran the project's tests and made sure they all passed. Projects with broken builds, with failing tests, or with fewer than 15 passing tests were discarded. These steps were applied repeatedly until we identified 20 projects, listed in Table 2. The columns in this table state the following characteristics for these projects:

- **LOC:** total lines of JavaScript/TypeScript in the source code of the project being analyzed (not including packages imported by the project, or test/compiled code).
- **#fun (async):** total number of functions in the project source code; the number between the parentheses gives the number of `async` functions.
- **#await (IO):** total number of await-expressions in the project source code; the number between parentheses gives the number that are I/O-dependent (as described in Section 4.7).
- **#test:** the number of tests associated with the project.
- **IO:** the I/O environment on which the reordered await expressions depend. Here, FS is the file system and NW is the network.
- **Brief description:** of the project (summarized from the repository's README file).

[15] File system I/O-related packages our test projects use: `fs`, `fs-admin`, `fs-extra`, `fs-tree-utils`, `fs-exists-cached`, `mock-fs`, `cspell-io`, `path-env`, and `tmp`.
[16] Network I/O-related packages our test projects use: `http`, `https`, `express`, `client`, `socks`, `puppeteer`.

Measuring run-time performance

To determine the impact of reordering await-expressions, we measure the execution time of those tests that execute at least one await-expression that was reordered. Tests that only execute unmodified code are not affected by our transformation, so their execution time is unaffected. We constructed a simple coverage tool that instruments the code to enable us to determine which tests are affected by the reordering of await-expressions.

Performance improvements are measured by comparing runtimes of each affected test before and after the reordering transformation. For our experiments, we ran the tests 50 times and calculated the average running time for each test over those 50 runs. This procedure was followed both for the original version of the project, and for the reordered version.

We took several steps to minimize potential bias or inconsistencies in our experimental results. First, we minimized contention for resources by running all experiments on a "quiet" machine where no other user programs are running. For our OS we chose Arch linux: as a bare-bones linux distribution, this minimizes competing resource use between the tests and the OS itself (since there are fewer processes running in the background than would be the case with most other OSs). We also configured each project's test runner so that tests are executed sequentially[17], removing the possibility for resource contention between tests.

During our initial experiments we observed that the first few runs of test suites for the file system dependent projects were always slower, and determined this was due to some files remaining in cache between test runs, reducing the time needed to read them as compared to the first runs that read them directly from disk. To prevent such effects from skewing the results of our experiments, we introduced a "warm-up" phase in which we ran the tests 5 times before taking performance measurements. We also decided to run the tests for the version with reorderings applied *before* the original version. Hence, if there is any caching bias resulting from the order of the experiments it would just make our results worse.

For network-dependent projects, we decided to focus on projects whose test suites can be run locally (i.e., on `localhost`) rather than over some remote server. This way, we avoid any bias from the random network latency present on real networks. This also has the effect of minimizing the effect of our reorderings: in the presence of slow network requests, we would expect the await reordering to have an enhanced positive effect on performance. In answering RQ3, we perform an experiment to explore this conjecture.

All experiments were conducted on a Thinkpad P43s with an Intel Core i7 processor and 32GB RAM.

5.2 RQ1 (Applicability)

To answer RQ1, we ran *ReSynchronizer* on each of the projects described in Table 2. Table 3 displays some metrics on the results, namely:

- **Awaits Reordered (%):** the absolute number of await-expressions reordered, with the parenthetical giving what fraction this is of the project's total I/O-dependent `await`s
- **Tests Affected (%):** the total number of affected tests (i.e., the number of tests that execute at least one reordered await-expression), with the parenthetical giving the percentage of the project's total tests this represents. For example: for the Kactus project there are 172 impacted tests, which is 21.5% of the 799 tests associated with the project.

[17] Some of the projects we tested relied on jest for their testing, while others used mocha. By default, jest runs tests concurrently, so we relied on its command-line argument `runInBand` to execute tests sequentially. This issue does not arise in the case of mocha, which runs tests sequentially by default.

Table 3 Number and percentage of `await`s reordered, per test project.

Project	Awaits Reordered (%)	Tests Affected (%)	Resync Time (s)
kactus	72 (6.0%)	172 (21.5%)	121
webdriverio	9 (7.1%)	12 (0.6%)	19
desktop	67 (5.4%)	187 (22.3%)	177
fiddle	3 (2.8%)	2 (0.3%)	8
nodemonorepo	22 (13.8%)	15 (3.0%)	7
zapier-platform-cli	16 (27.1%)	2 (5.6%)	5
wire-desktop	31 (13.1%)	14 (37.8%)	6
cspell	22 (9.7%)	26 (2.7%)	8
sourcecred	22 (11.5%)	29 (1.6%)	14
bit	116 (5.4%)	8 (2.0%)	204
vscode-psl	19 (4.7%)	116 (25.8%)	8
gatsby	103 (12.5%)	43 (1.6%)	30
jamserve	59 (5.5%)	272 (7.0%)	62
get	6 (20.7%)	3 (6.0%)	5
cucumber-js	13 (41.9%)	17 (3.1%)	64
sapper	35 (81.4%)	4 (2.6%)	26
svelte	5 (27.8%)	1 (0.03%)	67
reflect	4 (66.7%)	3 (18.8%)	12
m...-redux	3 (0.42%)	6 (0.45%)	85
enquirer	1 (6.7%)	71 (40.6%)	27

From this table, it can be seen that our analysis reorders between 0.4% and 81.4% of the I/O-dependent await-expressions (17.8% on average). While the number of reorderings strongly depends on the nature of the project being analyzed, it is clear that a nontrivial number of asynchronous computations has been scheduled suboptimally.

From the **Tests Affected** column in this table, it can be seen that between 0.03% and 40.6% of the projects' tests execute code affected by reorderings (9.4% on average), which is also a huge range. Note that the number of affected tests is not necessarily correlated with the number of `await`s reordered either: indeed, `cucumber-js`, the project with the highest fraction of `await`s reordered, has one of the lowest fractions of affected tests at only 3.1%. Clearly, the number of affected tests depends strongly on the way the developers structured their tests and on the distribution of the reorderings across the project. This underscores how important it is to only consider the affected tests when measuring the impact of the reorderings on performance, to avoid the results being skewed by unaffected tests.

5.3 RQ2 (Soundness)

The results in Table 3 demonstrated that *ReSynchronizer* was able to identify many await expressions that are candidates for reordering. However, if the unsoundness of the analysis would lead to many invalid reordering suggestions, the tool would not be very useful.

To determine if this unsoundness manifests itself in practice, we checked if the reorderings suggested by *ReSynchronizer* caused any test failures. In practice, we have not observed any situations where unsoundness manifests itself via invalid reorderings. In the 20 subject applications, we did not observe a single case where reordering await-expressions caused a test failure. While this is no guarantee that *ReSynchronizer* always proposes program behavior-preserving reorderings, it does suggest that the refactorings suggested by *ReSynchronizer* are not significantly less reliable than many state-of-the-art in refactoring tools.

Table 4 Results of performance experiments on github projects – Tests.

Project	Avg Speedup (%)	Max Speedup (%)	% Sig Speedup (%)
kactus	7.2%	32.4%	80.2%
webdriverio	1.5%	5.4%	16.7%
desktop	8.3%	35.4%	90.9%
fiddle	9.4%	16.6%	50.0%
nodemonorepo	3.5%	10.5%	86.7%
zapier-platform-cli	8.0%	8.9%	100.%
wire-desktop	5.4 %	17.3%	50.0%
cspell	4.3%	14.1%	50.0%
sourcecred	5.2%	20.2%	48.3%
bit	4.6%	16.7%	15.4%
vscode-psl	8.6%	75.0%	8.6%
gatsby	8.7%	52.2%	44.2%
jamserve	0.99%	23.1%	12.9%
get	1.3%	3.4%	33.3%
cucumber-js	12.3%	62.5%	17.6%
sapper	53.6%	80.1%	25.0%
svelte	6.8%	6.8%	100.%
reflect	1.1%	7.3%	66.7%
m...-redux	7.8%	9.2%	50.0%
enquirer	4.2%	38.1%	14.1%

5.4 RQ3 (Performance Impact)

Table 4 shows the results of our performance experiments, with the following columns:

- **Avg Speedup (%):** the average percentage speedup over all affected tests for the project. This is computed as $1 - harmean\left(\frac{t_i \text{ average time with reordering}}{t_i \text{ average time with original code}}\right)$; the harmonic mean[18] of this timing ratio over all affected tests t_i. If this value is negative it indicates a slowdown.
- **Max Speedup (%):** the maximum percentage speedup (i.e., the speedup for the test which was most improved by our reordering).
- **% Sig Speedup (%):** the percentage of tests for which there was a *statistically significant* speedup. We want to count how many of the tests were sped up by our reordering; but if we just counted how many tests had an average speedup after reordering, this would not account for the variance of our data. To address this, we performed a standard two-tailed t-test with the timings for each test with and without the reorderings. The t-test indicates a significant result only when the measured difference in timing is large with respect to the variability of the data, with "how large" being controlled by the confidence level (here, we chose 90% confidence). This is a measure of the proportion of the affected tests that our technique actually improved (with 90% confidence).
- Average run times (in seconds) for each individual affected test with and without reordering, for all projects, are included in the supplementary materials.

From Table 4, we see that the average speedups for the affected tests ranges from 0.99% to 53.6% for the projects under consideration, whereas maximum speedups range from 3.4% to 80.1%, suggesting that there is a large amount of variability in the performance improvements. As a result, one might wonder what effect these tests with huge improvements

[18] The harmonic mean is used since we are computing the average of ratios.

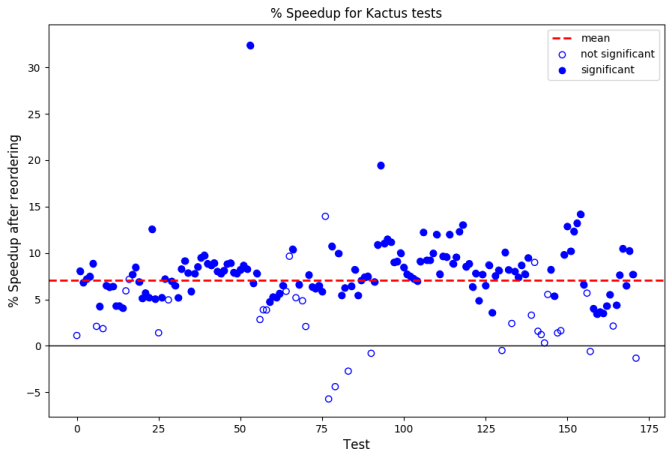


Figure 5 Average percentage speedups for all Kactus tests.

have on the average speedup, and whether a few outliers are significantly skewing the data. We address this with our last column, which shows the proportion of the tests for which we see a statistically significant speedup. Here too, we see a big range, with 8.6% to 100.% of the affected tests seeing statistically significant speedups.

To better understand the variability in our experimental results, we decided to take a closer look at the observed average speedups for all individual tests for the Kactus project[19], shown in Figure 5. This chart shows the percentage speedup as a result of reordering 72 await-expressions in Kactus, for each of Kactus's 172 impacted tests. Here, results for tests for which the reordering has a statistically significant effect on the runtime are depicted as colored circles, and those where the effect is not significant are shown as empty circles.

From Table 4 we recall that 80.2% of Kactus's affected tests are statistically significantly sped up, and indeed on this graph the vast majority of the tests experience a significant effect. From this graph we also get some information that is not available in the table: looking at the distribution of test speedups, we see that the test with the maximum speedup of 32.4% is indeed an outlier. We also see that most of the tests have speedups clustered fairly closely around the average of 7.2% (indicated by the dashed line on the graph). This is encouraging, as it means our reordering has a fairly consistent positive effect on the performance of Kactus. Finally, we see that although there are a few tests that incur a slowdown, none of these indicate a significant effect.

Prompted by these results, we decided to take an even closer look at the variability in our results. To this end, we created Figure 6, which shows the individual runtimes for each experiment run of one specific test of Kactus. For this, we chose as representative test #117, which executes the code in the motivating example presented in Section 3, and for which we observed an average speedup of 9.5%, which is fairly close to the mean of 7.2%. The figure displays the runtimes for this test both with the original version of Kactus and with the version with all reorderings applied. The mean of each of these runtimes is indicated using dot-dashed and dashed lines respectively.

[19] Supplemental materials include results from similar experiments with the other 19 subject applications.

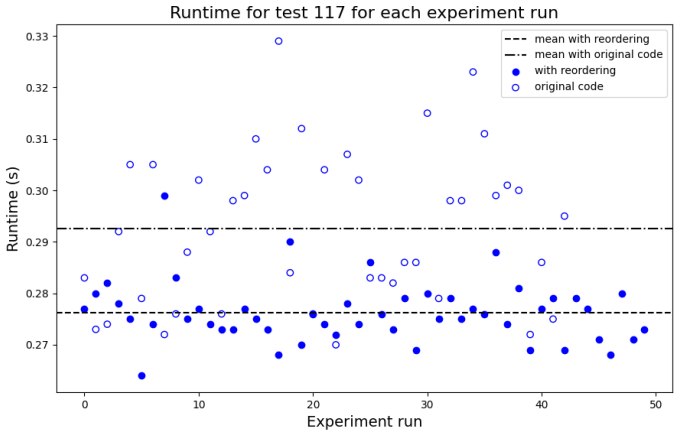


Figure 6 Runtimes (in seconds) for all experiment runs of Kactus test 117.

From Figure 6, we observe that there is *less variation in the running time of the test after reordering*. This same pattern is seen with other tests[20]. Our conjecture is that this reduction in variability of running times occurs because, before reordering, a test will experience the sum of the times needed to access multiple files, each of which may exhibit worst-case access time behavior. However, after reordering, when files are being accessed concurrently, the test execution experiences the maximum of these file-access times, i.e., experiencing the sum of the worst-case file access behaviors no longer occurs. We see the same phenomenon with network accesses[21]. This reduction in runtime variability is a positive side effect of the transformation, as it makes application runtime more stable and predictable.

To determine the impact of network latency on the performance of network-dependent reorderings, we conducted an experiment where we simulated different amounts of latency by manually[22] adding slowdowns of 50ms, 100ms, and 200ms to all the network calls that reordered await-expressions depend on. In each case, we ran the tests suites 50 times with and without the reordering, and report the average. Table 5 displays the results of this experiment. Generally, as network latency increases so too does the speedup due to the reordering. The only exception to this trend is seen as latency increases from 100ms to 200ms for the `reflect` project, where the average speedup goes from 2.9% to 2.8%. This small decrease is easily explained: with a big enough latency the runtimes are increased so that the relative difference from the speedup is smaller[23].

This is what we expected, since with the reordering multiple slow requests can be running at the same time and the execution does not need to wait for the total sum of all the latent request times. We also see that the percentage of affected tests where the speedup is significant either increases or is unchanged. From this experiment, we conclude that our reordering transformation becomes even more helpful as network latency increases.

[20] Supplementary materials include similar graphs for a few other tests, all of which follow the same trend.
[21] Supplementary materials include some graphs analogous to Figure 6 for network-dependent projects.
[22] To add the slowdowns, we follow the strategy used in the npm package `connect-slow`[3], which wraps a network call in a call to `setTimeout` using the specified slowdown time.
[23] E.g., for `reflect` test 1, we see average runtimes of 0.250s and 0.229s for 100ms latency (without/with reordering resp.), which is a speedup of 7.7%. Then, for 200ms latency the same test sees runtimes of 0.451s and 0.417s (without/with reordering resp), which only corresponds to a 6.2% speedup.

Table 5 Effect of await reorderings with and without simulated network latency.

	No Latency		50ms Latency		100ms Latency		200ms Latency	
Project	Avg	% Sig	Avg	% Sig	Avg	% Sig	Avg	% Sig
sapper	53.6%	25.0%	53.9%	25.0%	55.2%	75.0%	59.4%	75.0%
svelte	6.8%	100.%	7.9%	100.%	10.8%	100.%	11.8%	100.%
reflect	1.1%	66.7%	2.3%	66.7%	2.9%	66.7%	2.8%	66.7%
m...-redux	7.8%	50.0%	20.2%	100.%	20.3%	100.0%	22.3%	100.%
enquirer	4.2%	14.1%	7.7%	97.2%	18.3%	97.2%	35.0%	97.2%

5.5 RQ4 (Analysis Time)

Table 3's last column shows the time required by *ReSynchronizer* to process each of the subject projects, which range from 10k-160k lines of code. As can be seen from the table, the longest analysis time was 204 seconds. Applying the program transformation took less than 5 seconds for each project tested. Hence, our analysis scales to large applications.

5.6 Threats to Validity

Beyond the risks caused by the unsoundness of the static analysis that we already discussed, we consider the following threats to validity.

It is possible that the 20 projects used in our evaluation are not representative of JavaScript projects using `async`/`await`, so our results might not generalize beyond them. However, these projects were selected at random, and we observed the same trends among them.

In designing our performance evaluations, we were mindful of potential sources of bias to our results. We described the reasoning behind our design and how we mitigated bias in Section 5.1. In the case of caching bias, we ran our tests with reordered code *before* the tests for the original code, so that any bias would be against us.

Finally, our results might not generalize to I/O other than the file system or the network, such as database I/O. We conjecture that they will, as the logic of splitting an await-expression to maximize concurrency is environment-agnostic.

6 Related Work

This section covers related work on side-effect analysis and on refactorings related to asynchrony and concurrency.

Side-Effect Analysis

Our paper relies on interprocedural side-effect analysis to determine whether statements can be reordered without changing program behavior. Work on side-effect analysis started in the early 1970s, with the objective of computing dataflow facts that can be used to direct compiler optimizations.

Spillman[31] presents a side-effect analysis for the PL/I programming language that computes the expressions whose value may change as a result of assignments to variables. Spillman's analysis accounts for aliasing induced by pointers and parameter-passing, and is specified operationally as a procedure that creates a matrix associating variables with all expressions whose value would be impacted by an assignment to that variable. Procedure invocations are represented by additional rows in the matrix and side-effects for such invocations are computed in invocation order, using a fixpoint procedure to handle recursion.

A few years later, Allen[1] presents an interprocedural data flow analysis in which a simple intraprocedural analysis first identifies definitions that may affect uses outside a block, and uses in a block that may be affected by definitions outside the block. An interprocedural analysis then traverses a call graph in reverse invocation order to combine the facts computed for the individual procedures. Allen's algorithm does not handle recursive procedures.

Banning[4] presents an interprocedural side-effect analysis that accounts for parameter-induced aliasing in a language with nested procedures, and defines notions MOD and REF for flow-insensitive side-effects, and USE and DEF for flow-sensitive side-effects. Banning's flow-insensitive technique determines the set of variables immediately modified by a procedure and assumes the availability of a call graph to map variables in a callee to variables in a caller. The side-effect of a procedure call is then computed by way of a meet-over-all-paths solution. Our analysis follows Banning's approach but defines MOD and REF in terms of access paths [22] instead of names of variables, and relies on SSA form for improved precision (for access paths rooted in local variables).

Cooper and Kennedy[5] present a faster algorithm for solving the same problem of alias-free flow-insensitive side-effect analysis as Banning[4]. To improve the performance of the algorithm, they divide the problem into two distinct cases: side-effects to *reference parameters* (i.e., interprocedural function parameter aliasing), and *global variables*. They introduce a new data structure, the binding multigraph, for side-effect tracking through reference parameters, and a new, linear algorithm for side-effect tracking through global variables.

Later work by Landi et al. [17] focused on computing MOD sets for languages with general-purpose pointers. Pointers introduced another type of aliases to the problem of computing side effects, and Landi et al. extended previous work on computing MOD sets, by adapting and incorporating an existing algorithm for approximating pointer-based aliases.

Since their introduction by Banning[4], MOD and REF algorithms have also been adopted for use as parts of other dataflow analyses. Lapkowski and Hendren [18] present an algorithm for computing SSA numbering for languages with pointer indirection, which relies on MOD/REF side-effect analysis to track when the variable referred to by an SSA representation is being reassigned (in order to signal the need for a new SSA number).

Cytron et al.[6] also present an algorithm for computing SSA form which makes use of the MOD and REF side-effect analysis in order to determine when a variable could be modified indirectly by a statement. This work does not consider aliasing through pointers, and just uses the reference parameter and global variable aliasing as presented by Banning.

Refactorings related to Asynchrony and Concurrency

Gallaba et al.[11] present a refactoring for converting event-driven code into promise-based code. They assume that event-driven APIs conform to the error-first protocol (i.e., the first parameter of the callback functions is assumed to be a flag indicating whether an error occurred) and consider two strategies: "direct modification" and "wrap-around", where the latter approach is similar to "promisification" performed by libraries such as `universalify`. Their work predates the wide-spread adoption of `async`/`await` and does not show how to introduce these features, though there is a brief discussion how some of the presented mechanisms provide a first step towards refactorings for introducing `async`/`await`.

Dig[7] presented an overview of the challenges associated with refactorings related to the introduction and use of asynchronous programming features for Android and C# applications. Lin et al.[20] present *Asynchronizer*, a refactoring tool that enables developers to extract long-running Android operations into an `AsyncTask`. Since Java is multi-threaded, Android

applications may exhibit real concurrency, so (unlike with the JavaScript applications that we consider in our work) care must be taken to prevent data races that may cause nondeterministic failures. To this end, Lin et al. extend a previously developed static data race detector [26]. In later work, Lin and Dig[19] study the use of Android's three mechanisms for asynchronous programming: `AsyncTask`, `IntentService`, and `AsyncTaskLoader` and the scenarios for which each of these mechanisms is well-suited. They observe that developers commonly misuse `AsyncTask` for long-running tasks that it is not suitable for, and present a refactoring tool, *AsyncDroid*, that assists with the migration to `IntentService`.

Okur et al.[24] studied the use of asynchronous programming in C#, soon after that language added an `async`/`await` feature in 2012. At the time of this study, callback-based asynchronous programming was still dominant, although `async`/`await` was starting to be adopted widely. To facilitate the transition, Okur et al. created a refactoring tool, *Asyncifier* for automatically converting C# applications to use `async`/`await`. Okur et al. also observed several common anti-patterns involving the misuse of `async`/`await`, including unnecessary use of `async`/`await` and using long-running synchronous operations inside of `async` methods, and developed another tool, *Corrector* for detecting and fixing some of these issues.

Several other projects are concerned with refactorings for introducing and manipulating concurrency. Dig et al.[9] presented *Relooper*, a refactoring tool for converting sequential loops into parallel loops in Java programs. Wloka et al.[32] presented *Reentrancer*, a refactoring tool for making existing Java applications reentrant, so that they can be deployed on parallel machines without concurrency control. Dig et al.[8] presented *Concurrencer*, a refactoring tool that supports three refactorings for introducing AtomicInteger, ConcurrentHashMap, and FJTask data structures from the `java.util.concurrent` library. Okur et al.[23] presented two refactoring tools for C#, *Taskifier* and *Simplifier*, for transforming Thread and ThreadPool abstractions into Task abstractions, and for transforming Task abstractions into higher-level design patterns.

Schäfer et al.[29] present a framework of synchronization dependences that refactoring engines must respect in order to maintain the correctness of a number of commonly used refactorings in the presence of concurrency. Khatchadourian et al.[15] present a refactoring for migrating between sequential and parallel streams in Java 8 programs.

Kloos et al.[16] present JSDefer, a refactoring tool aimed at improving webpage performance by increasing concurrent loading of embedded scripts. This is done by deferring independent webpage scripts; like *ReSynchronizer*, JSDefer reasons about the dependence of their reordering targets in order to determine if the reordering will affect functionality. However, unlike our work, Kloos et al. make use of a *dynamic* analysis to determine dependence. JSDefer is also reordering entire scripts instead of individual statements.

7 Future Work

The main limitation of *ReSynchronizer* is the unsoundness and precision of the static analysis. Given the highly dynamic nature of JavaScript, this is hard to address, so one avenue of future work involves incorporating a *dynamic analysis* in *ReSynchronizer* to track data dependences between statements precisely. This would enable *ReSynchronizer* to perform additional reorderings by disregarding statements that "blocked" reordering due to being flagged as having global/environmental side effects by the static analysis. In particular, this is likely to help with calls to functions that are conservatively assumed to have global side effects such as `eval` and `setTimeout`. In our experience, these *often* do not actually have a data dependence with awaits being reordered, but static analysis is unable to determine that.

Relatedly, we are considering implementing an *interactive* usage mode. Here, the idea would be for *ReSynchronizer* to prompt the developer if it notices that it could do a better reordering if only it could prove that some statement has no global effects, and proceed with the reordering if the developer confirms that this is the case. In particular, this mode could suggest reorderings determined by the dynamic analysis that the static analysis deemed unsafe.

As the concept of splitting up and reordering components of an await-expression is not specific to JavaScript, we also consider the possibility of extending this work to other languages with the `async`/`await` construct. In particular, we conjecture that we could apply a similar approach to C#. In that setting, the static analysis could likely be made more effective by leveraging the static guarantees provided by the type system. However, C#'s multi-threading would pose additional challenges.

8 Conclusions

The changing landscape of asynchronous programming in JavaScript makes it all too easy for programmers to schedule asynchronous I/O operations suboptimally. In this paper, we show that refactoring I/O-related await-expressions can yield significant performance benefits. To identify situations where this refactoring can be applied, we rely on an interprocedural side-effect analysis that computes, for a statement s, sets $MOD(s)$ and $REF(s)$ of access paths that represent sets of memory locations modified and referenced by s, respectively. We implemented the analysis using CodeQL, and incorporated it into a tool, *ReSynchronizer*, that automatically applies the suggested refactorings. In an experimental evaluation, we applied *ReSynchronizer* to 20 open-source JavaScript applications that rely on file system or network I/O, and observe average speedups of between 0.99% and 53.6% (8.1% on average) when running tests that execute refactored code. While the analysis is potentially unsound, we did not encounter any situations where applying the refactoring causes test failures.

References

1. Frances E. Allen. Interprocedural data flow analysis. In Jack L. Rosenfeld, editor, *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, pages 398–402. North-Holland, 1974.
2. Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, pages 2:1–2:25, 2016. doi:10.4230/LIPIcs.ECOOP.2016.2.
3. Gleb Bahmutov. connect-slow. https://github.com/bahmutov/connect-slow, 2020. Accessed: 2020-12-13.
4. John Banning. An efficient way to find side effects of procedure calls and aliases of variables. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 29–41. ACM Press, 1979. doi:10.1145/567752.567756.
5. Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 57–66, 1988. doi:10.1145/53990.53996.
6. 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. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

7 Danny Dig. Refactoring for asynchronous execution on mobile devices. *IEEE Software*, 32(6):52–61, 2015. doi:10.1109/MS.2015.133.

8 Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 397–407, 2009. doi:10.1109/ICSE.2009.5070539.

9 Danny Dig, Mihai Tarce, Cosmin Radoi, Marius Minea, and Ralph E. Johnson. Relooper: refactoring for loop parallelism in Java. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 793–794, 2009. doi:10.1145/1639950.1640018.

10 ECMA. Ecmascript 2019 language specification, 2010. Available from http://www.ecma-international.org/ecma-262/.

11 Keheliya Gallaba, Quinn Hanam, Ali Mesbah, and Ivan Beschastnikh. Refactoring asynchrony in JavaScript. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 353–363. IEEE Computer Society, 2017. doi:10.1109/ICSME.2017.83.

12 GitHub. CodeQL. https://github.com/codeql, 2021. Accessed: 2021-01-05.

13 GitHub. CodeQL standard libraries and queries. https://github.com/github/codeql, 2021. Accessed: 2021-01-05.

14 Jordan Harband. util.promisify. https://github.com/ljharb/util.promisify, 2020. Accessed: 2020-05-14.

15 Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. Safe automated refactoring for intelligent parallelization of java 8 streams. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 619–630. IEEE / ACM, 2019. doi:10.1109/ICSE.2019.00072.

16 Johannes Kloos, Rupak Majumdar, and Frank McCabe. Deferrability analysis for JavaScript. In *Haifa Verification Conference*, pages 35–50. Springer, 2017.

17 William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *In Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, 1993.

18 Christopher Lapkowski and Laurie J Hendren. Extended ssa numbering: Introducing SSA properties to languages with multi-level pointers. In *International Conference on Compiler Construction*, pages 128–143. Springer, 1998.

19 Yu Lin, Semih Okur, and Danny Dig. Study and refactoring of Android asynchronous programming (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 224–235, 2015. doi:10.1109/ASE.2015.50.

20 Yu Lin, Cosmin Radoi, and Danny Dig. Retrofitting concurrency for Android applications through refactoring. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 341–352, 2014. doi:10.1145/2635868.2635903.

21 Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static Analysis of Event-Driven Node.js JavaScript Applications. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2015.

22 Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPIcs*, pages 7:1–7:24. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

23 Semih Okur, Cansu Erdogan, and Danny Dig. Converting parallel code from low-level abstractions to higher-level abstractions. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pages 515–540, 2014. doi:10.1007/978-3-662-44202-9_21.

24 Semih Okur, David L. Hartveld, Danny Dig, and Arie van Deursen. A study and toolkit for asynchronous programming in C#. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1117–1127, 2014. doi:10.1145/2568225.2568309.

25 pandas. pandas. https://pandas.pydata.org, 2020. Accessed: 2020-12-13.

26 Cosmin Radoi and Danny Dig. Practical static race detection for Java parallel loops. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 178–190, 2013. doi:10.1145/2483760.2483765.

27 Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *ACM SIGPLAN Notices*, volume 48, pages 151–166. ACM, 2013.

28 Max Schäfer and Oege de Moor. Specifying and implementing refactorings. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 286–301. ACM, 2010. doi:10.1145/1869459.1869485.

29 Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. Correct refactoring of concurrent Java code. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 225–249, 2010. doi:10.1007/978-3-642-14107-2_11.

30 Isaac Z. Schlueter. graceful-fs. https://www.npmjs.com/package/graceful-fs, 2020. Accessed: 2020-05-14.

31 Thomas C. Spillman. Exposing side-effects in a PL/I optimizing compiler. In *Information Processing, Proceedings of IFIP Congress 1971, Volume 1 - Foundations and Systems, Ljubljana, Yugoslavia, August 23-28, 1971*, pages 376–381, 1971.

32 Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for reentrancy. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 173–182, 2009. doi:10.1145/1595696.1595723.

33 Ryan Zim. universalify. https://github.com/RyanZim/universalify, 2020. Accessed: 2020-05-14.

Differential Privacy for Coverage Analysis of Software Traces

Yu Hao
Ohio State University, Columbus, OH, USA

Sufian Latif
Ohio State University, Columbus, OH, USA

Hailong Zhang
Fordham University, New York, NY, USA

Raef Bassily
Ohio State University, Columbus, OH, USA

Atanas Rountev
Ohio State University, Columbus, OH, USA

Abstract

This work considers software execution traces, where a trace is a sequence of run-time events. Each user of a software system collects the set of traces covered by her execution of the software, and reports this set to an analysis server. Our goal is to report the local data of each user in a *privacy-preserving manner* by employing local differential privacy, a powerful theoretical framework for designing privacy-preserving data analysis. A significant advantage of such analysis is that it offers principled "built-in" privacy with clearly-defined and quantifiable privacy protections. In local differential privacy, the data of an individual user is modified using a *local randomizer* before being sent to the untrusted analysis server. Based on the randomized information from all users, the analysis server computes, for each trace, an estimate of how many users have covered it.

Such analysis requires that the domain of possible traces be defined ahead of time. Unlike in prior related work, here the domain is either infinite or, at best, restricted to many billions of elements. Further, the traces in this domain typically have structure defined by the static properties of the software. To capture these novel aspects, we define the trace domain with the help of context-free grammars. We illustrate this approach with two exemplars: a *call chain analysis* in which traces are described through a regular language, and an *enter/exit trace analysis* in which traces are described by a balanced-parentheses context-free language. Randomization over such domains is challenging due to their large size, which makes it impossible to use prior randomization techniques. To solve this problem, we propose to use *count sketch*, a fixed-size hashing data structure for summarizing frequent items. We develop a version of count sketch for trace analysis and demonstrate its suitability for software execution data. In addition, instead of randomizing separately each contribution to the sketch, we develop a much-faster one-shot randomization of the accumulated sketch data.

One important client of the collected information is the identification of high-frequency ("hot") traces. We develop a novel approach to identify hot traces from the collected randomized sketches. A key insight is that the very large domain of possible traces can be efficiently explored for hot traces by using the frequency estimates of a visited trace and its prefixes and suffixes. Our experimental study of both call chain analysis and enter/exit trace analysis indicates that the frequency estimates, as well as the identification of hot traces, achieve high accuracy and high privacy.

2012 ACM Subject Classification Software and its engineering → Dynamic analysis; Security and privacy → Privacy-preserving protocols

Keywords and phrases Trace Profiling, Differential Privacy, Program Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.8

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.7`

Funding This material is based upon work supported by the National Science Foundation under Grant No. CCF-1907715.

Acknowledgements We thank the ECOOP reviewers for their valuable feedback.

1 Introduction

In this work we consider privacy-preserving coverage analysis for software traces. A trace is an event sequence $t \in \mathcal{E}^+$ over some pre-defined set of possible run-time events $\mathcal{E}$. Consider a software system deployed over a set of software users $u_1, \ldots, u_n$. Each user u_i executes her own copy of the software and that execution records a local set T_i of traces that were observed at run time. This data is collected locally and then sent to an analysis server. We consider the following problem: for each $t \in \mathcal{E}^+$, estimate the frequency of t over the population of users, that is, $f(t) = |\{i : t \in T_i\}|$, *while collecting the local data of each user in a privacy-preserving manner and sending it to an untrusted analysis server.*

Trace information has a wide range of uses in software analysis and transformation. For example, high-frequency traces can focus software optimization, testing, and static checking on important user behaviors. Similarly, behavior flow analysis in mobile/web analytics [21, 19, 22, 37] allows developers to optimize the functionality and performance of common paths taken by app users through the app code.

1.1 The Need for Privacy-Preserving Analysis

Our goal is to design an analysis that obtains accurate trace coverage statistics across a population of software users, while controlling carefully the "privacy loss" of each user. This is motivated by the increased importance of reducing the amount of user information collected by business entities. Both societal and legislative pressures have highlighted the need for such reduction. For software-generated event information – for example, collected with the help of popular analysis infrastructures for mobile/web analytics (e.g., provided by Google and Facebook) – typically there are no "built-in" privacy protection mechanisms. The infrastructures themselves collect a wealth of information, including user IP addresses and GUI events. App-specific data collection can provide even more fine-grained knowledge about user's behavior and interaction with the software. For example, trace coverage information can provide details about what paths through the code a user has taken, and what functionality (possibly sensitive) she has executed. This data could potentially be used to infer user-specific habits, interests, and characteristics.

From the point of view of software users, the release of data collected from software executions is often undeclared or obscured. Even if the user is aware of the data collection, they are unlikely to have true appreciation of its implications. What is particularly troubling is that the collected data could be linked with other sources of information about this user (which cannot be prevented even with anonymization [34, 35]) and could be used as part of future larger-scale data mining and machine learning attempts to infer user-specific information. At data collection time, it is impossible to predict what extra data sources will be linked and what future inferences will be possible using that data.

Privacy-preserving data analysis aims to develop systematic mechanisms for addressing this problem. Such analysis benefits two categories of stakeholders. First, the privacy of individual users is protected in a well-defined manner. Further, entities performing data collection (e.g., Google and app developers using Google's analytics infrastructure) benefit as well: they are responsive to privacy expectations and do not have access to raw data that can compromised by unexpected data leaks or unethical business practices. In this work we focus on one particular privacy-preserving mechanism: *local differential privacy*. Our goal is to use local differential privacy to design a privacy-preserving trace coverage analysis.

1.2 Local Differential Privacy

Differential privacy [16] is a powerful theoretical framework for privacy-preserving data analyses. This approach has been recognized by the theoretical computer science community [1] and has been employed by high-profile companies such as Google, Apple, and Microsoft [18, 4, 33, 51] and by the U.S. government [14]. More widespread use of this technology is becoming possible via recent open-source tools [40].

A significant advantage of differentially-private data analyses is that they offer principled "built-in" privacy with clearly-defined and quantifiable privacy protections. Broadly, privacy is achieved by randomizing the collected data in a way that ensures the impossibility of high-confidence inferences from the randomized data. There is increasing interest in using such techniques in the context of software. For example, the programming languages community has considered techniques for verification of differentially-private algorithms [56, 53, 36, 61]. Others have considered uses of differential privacy for event frequency analysis of deployed software [58, 60, 57], with natural applications to mobile app analytics [59].

In local differential privacy, the data of an individual user is modified using a *local randomizer* before being sent to an untrusted analysis server. Thus, the raw local information never leaves the user's environment. In our setting the analysis server computes, for each trace, an estimate of how many users have covered it.

A major advantage of this approach is that it is resilient to other known and unknown categories of knowledge that may become available about this user in the future. Thus, a differentially-private analysis is designed to be robust even under the current trend of fast-growing data collection and linking from various sources, used by businesses and governments to create user behavior profiles and to mine them for user-specific patterns [55, 16]. Such user protection is also important under the threat of unexpected data releases caused by obscure changes to privacy agreements, information requests by law enforcement, or security breaches in which user data is compromised. The quantifiable privacy-preserving machinery of local differential privacy is appealing not only to software users, but also to software developers and analysis infrastructure providers. Both the developers and the infrastructure providers can claim, with confidence, that they deploy privacy-by-design in their data collection. As a result, the data they collect and store is protected, in the statistical differentially-private sense, against data breaches, rogue employees/subcontractors, and scrutiny from government agencies and law enforcement. Section 2 further discusses the assumptions behind this analysis model and the nature of privacy protection achieved under it.

1.3 Challenges and Contributions

Various differentially-private approaches have been considered for similar styles of problems. Some have considered data analysis with single data item per user [18, 52, 8]. Others have studied data collection for a pre-defined small set of items from software executions [58, 57, 60, 59]. However, the analysis of software traces presents unique challenges. Our contributions in solving these challenges are summarized below.

Contribution 1: Analysis for structured large domains

Differentially-private analyses require that the domain of possible data items be defined ahead of time, as part of algorithm design. Unlike prior related work where software executions generate data over a small unstructured domain containing a few thousand elements [58, 57, 60], here the domain is either infinite or, at best, restricted to many billions of elements. Further, the traces in this domain typically have structure defined by the static properties of the software. To capture these novel aspects, we propose to define the trace

domain with the help of context-free grammars. This approach has the key advantage that both the domain definition and the exploration of its elements are formulated using popular programming language machinery. We illustrate this approach with two exemplar analyses: a *call chain analysis* in which traces are described through a regular language, and an *enter/exit trace analysis* in which traces are described by a balanced-parentheses context-free language. Both kinds of structures are widely used in program analysis and are applicable to a range of techniques beyond these two exemplars. This formulation plays a key role in our approach for identification of high-frequency domain elements, as described later.

Contribution 2: Count sketch

The domain of possible traces is very large. For example, in a realistic setting for our benchmarks, this set has billions of elements. One of the key features of a differentially-private approach is that it produces per-user randomized data *that could contain an arbitrarily large subset* of this extremely large domain. This is essential for achieving the differential privacy guarantee, but is clearly impractical for our purposes, in terms of both space and generation time. We address this exponential explosion by using *count sketch* [10], a fixed-size hashing data structure originally designed to collect data about frequent items in data streams. While prior work [8] has considered the theoretical applications of count sketch for a simple single-item differentially-private data analysis, we develop a version of count sketch that is applicable to the more complex analysis we consider, and demonstrate its suitability for real software execution data.

Contribution 3: Efficient randomization

The standard approach for designing the local randomizer is to randomize individual contributions (i.e., observations of covered traces) as they are observed. However, our experience shows that the cost of such randomization is high and not suitable for real-world software analysis. Instead, we develop a technique to accumulate the effects of unrandomized count sketch updates, and only perform the local randomization as a one-shot step on this accumulated data. This reduces the cost of the randomization by orders of magnitude and makes it practical to use for realistic data gathering from deployed software.

Contribution 4: Analysis of hot traces

After the randomized local data is collected by the analysis server, a resulting *global count sketch* captures the population-wide information about observed traces. From this global sketch, a frequency estimate can be obtained for any given trace t from the domain of possible traces. However, this alone is not enough for many forms of data analyses, since the number of possible traces is exponential or even infinite, and obtaining an estimate for each t (and then analyzing all these estimates) is not possible. We focus on one particular data analysis of significant importance: identifying the *hot traces* and estimating their frequencies. A hot trace has a frequency that exceeds some threshold. Knowledge of such traces is useful to identify common user behaviors, leading to focused performance optimization, testing/checking, and application-flow optimization. We develop a novel approach to identify likely-hot traces from the randomized data in the global sketch. The key insight of our approach is that the explosively-large domain of possible traces can be efficiently explored for hot traces by using the frequency estimates of a visited trace and its prefixes and suffixes. We illustrate this approach with the help of the two exemplar analyses mentioned earlier, and demonstrate how the exploration of the domain can be performed by exploring the states of the automaton corresponding to the underlying context-free language.

Contribution 5: Study of privacy/accuracy trade-offs

The trade-offs between accuracy and privacy are central in the design of privacy-preserving algorithms. Our experimental study characterizes these trade-offs in several dimensions for both call chain analysis and enter/exit trace analysis. The conclusions from this study can be summarized as follows: (1) frequency estimates for software traces can be obtained with both high accuracy and high privacy, as long as the data collection includes a sufficiently-large number of software users, (2) the set of hot traces can be discovered with high recall and precision by reasoning about a trace's prefixes and suffixes, (3) the frequency estimates for hot traces are accurate and better than the estimates for the remaining traces, (4) after a certain point in the accuracy/privacy trade-off spectrum, reduced privacy does not result in significantly better accuracy; this point provides a natural choice for selecting such trade-offs.

2 Background and Problem Statement

2.1 Software Traces

We consider software traces, collected over a set of software users u_i for $i \in [1, n]$. Each user u_i executes her own copy of the software. During execution, run-time events are observed and recorded. Let $\mathcal{E}$ be the finite set of possible run-time events. This set is defined before software deployment, as part of the design of the trace analysis. For convenience of definitions, we assume that $\mathcal{E}$ contains an artificial "start" event s denoting the start of a trace. A trace t is a string $t \in \mathcal{E}^+$, starting with s. We will use the notation $t = \langle s, e_1, \ldots, e_k \rangle$ to denote a trace t of length k. (Note that we exclude s when defining trace length.)

Let $\mathcal{T}$ be a domain describing conservatively (i.e., over-approximating) the set of all possible traces that could be observed at run time. We expect this domain to be statically described as part of the design of the trace analysis. In the simplest case, $\mathcal{T} = \mathcal{E}^+$. More precise definitions of $\mathcal{T}$ may be possible via static analysis. Regardless of the means to derive $\mathcal{T}$, we expect it to be very large (e.g., exponential in the static size of the program). In addition, traces typically have structure that is constrained by the static properties of the software. In particular, one important special case we investigate is when $\mathcal{T}$ is defined inductively through a family of "extension" functions ext_k: $\mathcal{E}^k \times \mathcal{E} \to \mathcal{P}(\mathcal{E}^{k+1})$. Here $\mathcal{P}(X)$ denotes the power set of X and k ranges over the natural numbers. For any $t \in \mathcal{T}$ of length k, $\mathsf{ext}_k(t)$ is the set of all traces $t' \in \mathcal{T}$ of length $k+1$ such that t is a prefix of t'. That is, $\mathsf{ext}_k(t)$ shows all ways in which t could be extended with one more event to form a valid trace. For simplicity, we will omit the subscript k in ext_k when it is clear from the context. As discussed later, this definition of $\mathcal{T}$ enables incremental search for "hot" traces. While our definitions of privacy-preserving analysis are conceptually applicable to broader categories of $\mathcal{T}$, the application of this approach for identification of traces that appear frequently in the user population requires such incremental definition of $\mathcal{T}$ (Section 4).

Below we discuss two examples of such trace domains $\mathcal{T}$, both with direct connections to popular categories of analyses. These exemplars illustrate how common properties of such analyses can be mapped to the problem definition and solution described in this work. In particular, we define these two domains via well-understood formal languages – a regular language and a balanced-parentheses context-free language – which provides a natural definition for the domain and its extension function. This choice is also motivated by the fact that such languages are widely used in various existing software analysis techniques. Our approach is directly applicable to other trace analyses where the trace domain has a similar structure. This machinery is likely to be generalizable to broader domains (e.g., ones that correspond to more general context-free languages) but we do not explore these generalizations in this work.

Both domains are based on a set of events corresponding to entering or exiting a software component (e.g., method, module, or GUI window). We simplify the definition by assuming that each component is uniquely identified by an integer id from $[1, c]$. In addition, we introduce an artificial component with id 0 which corresponds to the external environment – e.g., the caller of the main method, or the framework code that invokes Android app entry points. The set of events is then $\mathcal{E} = \mathit{Enter} \cup \mathit{Exit}$ where $\mathit{Enter} = \{\mathsf{enter}(i) : i \in [0, c]\}$ and $\mathit{Exit} = \{\mathsf{exit}(i) : i \in [0, c]\}$. The artificial start event s is $\mathsf{enter}(0)$.

2.1.1 Exemplar 1: Call Chains

We first describe an exemplar analysis in which the static domain $\mathcal{T}$ of possible traces is defined by a simple regular language. Suppose that we are given a set of static *call edges* $i \to j$ showing that, at run time, the execution of component i may trigger the execution of component j. A finite sequence $i \to j \to k \to \ldots$ of such call edges is a static *call chain*. A call chain denotes a trace of events "i calls j which in turn calls k which in turn calls $\ldots$". Equivalently, we can define the domain $\mathcal{T}$ through a regular language containing strings $t = \langle \mathsf{enter}(0), \mathsf{enter}(i_1), \ldots, \mathsf{enter}(i_k) \rangle$ over the alphabet *Enter*. The static call graph can be thought of as the finite-state automaton that defines this language, and the trace extension function ext is the transition function of that automaton.

2.1.2 Exemplar 2: Enter/Exit Traces

Next we define an exemplar analysis in which $\mathcal{T}$ is based on a balanced-parentheses context-free language. This language captures the standard notion of *interprocedurally valid paths* [46] and is defined by the following grammar:

$$\begin{aligned} \mathit{Valid} &\to \mathsf{enter}(i)\ \mathit{Valid} \mid \mathit{Balanced}\ \mathit{Valid} \mid \lambda \\ \mathit{Balanced} &\to \mathsf{enter}(i)\ \mathit{Balanced}\ \mathsf{exit}(i) \mid \mathit{Balanced}\ \mathit{Balanced} \mid \lambda \end{aligned}$$

where λ is the empty string. Non-terminal *Balanced* defines a sequence of matching enter and exit events. Starting non-terminal *Valid* describes a sequence with some not-yet-matched enter events. Grammars of similar structure have been used extensively in a wide variety of static analyses (e.g., [46, 48]). For our exemplar analysis we consider the domain of enter/exit traces $\mathcal{T}$ to be strings derived from *Valid* and starting with $\mathsf{enter}(0)$. We further restrict the strings to respect a given set of static call edges $i \to j$. This can be easily encoded in the definition of the corresponding pushdown automaton, as follows. We can define a deterministic pushdown automaton with a single state. The input alphabet is $\mathit{Enter} \cup \mathit{Exit}$ and the stack alphabet is *Enter*, with initial stack symbol $\mathsf{enter}(0)$. The transitions upon observing input event $\mathsf{enter}(j)$ when the top of the stack is $\mathsf{enter}(i)$ is defined only if there is a static call edge $i \to j$. This transition pushes $\mathsf{enter}(j)$ onto the stack. If the input symbol is $\mathsf{exit}(i)$, the transition is defined only if the top of the stack is $\mathsf{enter}(i)$, in which case the stack top is popped. The trace extension function ext, which captures all ways in which a given trace is extended with one more event, is easily derivable from this pushdown automaton.

There are two reasons we use these formalisms to describe our exemplar analyses. First, the underlying structure, defined by a finite-state automaton or a balanced-parentheses pushdown automaton, is commonly observed in a variety other of dynamic analyses. Our machinery can be directly employed for such analyses. Second, the automata naturally provide the definition of incremental algorithms to explore the domain of possible traces via the extension functions ext. As described later, such incremental algorithms play an important role in our identification of frequently-occurring domain elements. It may be

possible to generalize such machinery to more general pushdown automata, but the definition of the extension functions (derived from the automata) would be more complex than the simple extension functions described above.

2.2 Trace Coverage Analysis for Deployed Software

When the program is executed by a software user, some subset of $\mathcal{T}$ is actually observed (i.e., covered) at run time. A variety of run-time techniques can be used to determine this coverage (e.g., [7, 3, 62, 49]). We consider such coverage across a large number of software users, each running her copy of the program. Let there be n software users denoted by $u_1, \ldots, u_n$ and let $T_i \subseteq \mathcal{T}$ be the set of traces covered when user u_i executes the program. We consider the following *trace coverage analysis*: for each $t \in \mathcal{T}$, estimate the frequency of t over the population of users, that is, $f(t) = |\{i : t \in T_i\}|$, while collecting the local data of each user with differential privacy.

Trace information has been used extensively to analyze and optimize software performance [7, 3, 5, 62, 26, 2]. The frequency information defined above can be used to focus such efforts on important user behaviors. Similarly, testing and static checking can be focused on traces that are commonly observed in the user population. Another example is behavior flow analysis in mobile and web analytics frameworks [21, 37], which allows developers to see different paths that users take through the app. The paths can be thought of as a form of traces across GUI components, and the analysis annotates each edge with the number of users who have performed the corresponding transition. A similar example is funnel analysis [21, 19, 22, 37], which visualizes the completion rate of a task in terms of a series of specific events and helps developers find optimizations in their software design. Traces collected for funnel analysis may contain sensitive information. For example, events in trace "launch the app, open the news page, navigate to sports news, perform sports merchandise purchase" can be used for targeted advertising. Our approach allows developers to conduct frequency analysis while ensuring worst-case privacy guarantees even when users are unaware of the data being collected and the unexpected/unpredictable future uses of this data.

2.3 Differential Privacy

Differential privacy is applicable to data analyses where data is being collected from many participants, and some processing of this data produces results that are then made available to untrusted parties. Such untrusted parties could be, for example, government agencies and business entities. Two main models of differential privacy have been considered [16]. In the *centralized* model, a trusted "data curator" collects the raw data from participants, performs the data analysis, and releases the results to untrusted entities. As part of the data analysis, some form of randomization is applied to ensure the differential privacy guarantee (this guarantee will be described shortly). In the *local* model, the randomization is performed by each participant, and the resulting modified data is then released to untrusted entities, which perform data analysis on this data. Again, the randomization ensures the differential privacy guarantee. Our work focuses on the second scenario, which is well suited for analysis of deployed software. In the specific problem we consider, the raw data for software user u_i is the set T_i of locally-covered traces. The user applies a local randomizer R to this data and then reports $R(T_i)$. We assume a typical setting where the reported data is collected by an untrusted analysis server. This server analyzes the data from all users and computes estimates $\hat{f}(t)$ of the true frequencies $f(t)$.

Differential privacy guarantee

Suppose $R(T_i)$ is released publicly by a software user. We aim to design a randomizer R that ensures the following differential privacy property: *for every possible $t \in \mathcal{T}$, an external observer of $R(T_i)$ cannot have high confidence that the hidden raw data contains that t*. In other words, whether t is in T_i cannot be ascertained with high probability based only on the observation of $R(T_i)$. In essence, the presence of t in the private local data is hidden in a probabilistic sense.

More precisely, let $P[R(X) = Z]$ be the probability that given input X, the randomizer produces output Z. For any Z and any two $X \subseteq \mathcal{T}$ and $Y \subseteq \mathcal{T}$ that differ by a single element t, the ratio of $P[R(X) = Z]$ and $P[R(Y) = Z]$ should be bounded by e^{ϵ}. Here X and Y are considered to be "neighbors" in the space of inputs to the randomization algorithm. Because the two probabilities are close to each other, when someone observes any output Z, she cannot have much higher confidence in the statement "the raw data contained t", compared to the confidence she can have in the statement "the raw data did not contain t". Here ϵ is the *privacy loss parameter*, which is used to tune accuracy/privacy trade-offs. A typical value used in related work is $\ln(9)$ [18, 52, 58]; for example, this value is used in the "basic one-time" version of a popular randomization technique [18]. Larger values of ϵ improve the accuracy of analysis results, but weaken the privacy guarantee.

A key assumption is that the adversarial observer of $R(T_i)$ knows fully all details of how randomizer R works, for example, because this observer designed the randomizer in the first place, or because she reverse-engineered it from the program code. As part of this assumption, the observer also knows the value ϵ which was embedded in the randomizer design. Even under such strong assumptions, the differential privacy guarantee makes it impossible to distinguish, in a probabilistic sense, neighbor inputs to the randomizer after the randomizer output is publicly released. Such principled and quantifiable protection is one of the reasons differential privacy has been employed by companies such as Google [18], Microsoft [33], Apple [4], and Uber [51], as well as by the U.S. Census Bureau [14]. More widespread use of such protection has become possible via recent open-source tools for differentially-private analysis [40].

Randomized response

To illustrate this key *indistinguishability property*, we present a classic simplified example. For illustration, suppose that the raw data for user u_i is a single trace $t_i \in \mathcal{T}$. A well-know randomization technique is derived from *randomized response*, an approach used in social sciences to handle evasive answers to sensitive questions [54]. The randomizer $R : \mathcal{T} \to \mathcal{P}(\mathcal{T})$ takes as input a single trace t and produces a set of traces, based on the following rules: (1) the input t is included in the output with some probability p, and (2) for every other $t' \in \mathcal{T}$, t' is included in the output with probability $1 - p$. Thus, the real trace could be missing from the output, and any other trace could be part of the output. Note that this approach is applicable only when $\mathcal{T}$ is finite and, practically, the size of $\mathcal{T}$ is relatively small.

By selecting $p = e^{\frac{\epsilon}{2}}/(1 + e^{\frac{\epsilon}{2}})$, this approach provably achieves ϵ-indistinguishability: for any set $Z \subseteq \mathcal{T}$ and any two traces $t' \in \mathcal{T}$ and $t'' \in \mathcal{T}$, the probabilities $P[R(t') = Z]$ and $P[R(t'') = Z]$ can differ by at most a factor of e^{ϵ}. In other words, observing Z means that (1) the raw data that produced Z could have been any trace from $\mathcal{T}$, and (2) no trace from $\mathcal{T}$ is much more likely to have been the input, compared to the remaining elements of $\mathcal{T}$.

In this simplified problem, each user u_i reports $R(t_i)$ to the analysis server; here $1 \leq i \leq n$. The server produces estimates $\hat{f}(t)$ by computing $h(t) = |\{i : t \in R(t_i)\}|$ and then calibrating it in order to create an unbiased $f(t)$ estimate: $\hat{f}(t) = ((1 + e^{\frac{\epsilon}{2}})h(t) - n)/(e^{\frac{\epsilon}{2}} - 1)$.

2.4 Assumptions

Several assumptions need to be explicitly stated before we describe our differentially-private analysis (Section 3). As usual in this type of work, it is assumed that the design and implementation of the approach are fixed before any data collection and are publicly known by all stakeholders, including untrustred parties. Another assumption is that the software code correctly implements the design; in particular, it does implement the randomization as publicly announced, and does not try to circumvent it by sending the raw data (or some version of it) to a malicious party. Although this is a strong assumption, it is no different than what is currently used in remote analysis of deployed software, where the design is typically undocumented and/or obfuscated, and there is no checking of the implementation of the data collection for correctness or presence of malicious code.

If a software developer commits to using the correct design and implementing it as expected, this raises the confidence of software users and watchdog agencies that indeed privacy is protected. Further, several techniques can be used to increase this confidence, including (1) open-source implementations, (2) use of certified and trusted third-party libraries, (3) scrutiny by privacy experts, and (4) code analysis via automated tools. Note that there are no assumptions about the analysis server to which the randomized data is sent. This server could be part of a privacy attack, possibly involving additional external sources of information about the targeted software user. Even with this assumption, the differential privacy guarantee holds [55].

3 Randomized Count Sketch for Software Traces

Even if a user's local information contains a single trace, the approach outlined in the previous section is not possible when $\mathcal{T}$ is infinite, since every elements of $\mathcal{T}$ must be visited when randomization is applied. Even if $\mathcal{T}$ is made finite – for example, by using a pre-defined limit on trace length – the approach is still not practical. For illustration, consider call chains for the `localtv` Android app used in our experiments. The alphabet size $|Enter| = 2974$ in this app is close to the median for our set of benchmarks. Even if we only consider chains of at most three methods and count the strings recognized by the corresponding finite-state automaton (as described in Section 2.1.1), we have $|\mathcal{T}| = 3,272,137$. Increasing this length by one, the size of $\mathcal{T}$ becomes more than 163 million. A further length increase by one results in $|\mathcal{T}|$ of over 8 billion. Our implementation of the finite-state automaton is based on a call graph constructed through class hierarchy analysis. Using a more precise call graph analysis (e.g., based on context-sensitive analysis) may reduce $|\mathcal{T}|$. However, it is likely that $\mathcal{T}$ will still be very large, since conditional behaviors (e.g., calls guarded by conditionals) are common and easily produce exponential growth in the number of statically-possible traces. The cost of the randomizer described earlier is proportional to the size of $\mathcal{T}$, as each element $t \in \mathcal{T}$ must be visited and a random value must be generated for that t (independently of the processing of the remaining elements of $\mathcal{T}$) in order to decide whether t is included in the randomizer output. Further, the randomizer output, which needs to be sent to the analysis server, has size dependent on the exponentially-large size of $\mathcal{T}$. Clearly, these costs are infeasible.

3.1 Count Sketch

To address this problem we employ *count sketch* [10], a data structure originally designed to find frequent items in data streams. Prior work [8] has considered the theoretical analysis of using count sketch for a restricted form of differentially-private data analysis, where

Chain	$h_1(t), g_1(t)$	$h_2(t), g_2(t)$	$h_3(t), g_3(t)$
$t_1 = \langle 0, 473 \rangle$	$3, -1$	$6, -1$	$8, 1$
$t_2 = \langle 0, 93 \rangle$	$1, 1$	$7, 1$	$3, -1$
$t_3 = \langle 0, 473, 83 \rangle$	$5, -1$	$1, -1$	$4, -1$
$t_4 = \langle 0, 473, 472 \rangle$	$5, -1$	$4, 1$	$4, -1$
$t_5 = \langle 0, 473, 83, 1605 \rangle$	$5, 1$	$5, -1$	$2, 1$
$t_6 = \langle 0, 473, 472, 971 \rangle$	$8, 1$	$1, -1$	$7, 1$
$t_7 = \langle 0, 473, 472, 973 \rangle$	$7, -1$	$3, -1$	$4, 1$

Local Sketch

1	0	-1	0	-1	0	-1	1
-2	0	-1	1	-1	-1	1	0
0	1	-1	-1	0	0	1	1

Figure 1 Count sketch illustration, with $m = 8$ and $s = 3$.

each user has a single data item. However, there is no clarity on the practical use of this data structure for analysis of real-world software execution data and for the more general problem we consider, where each user has a set of local traces. Using insights from this prior work, we develop a version of count sketch for our trace analysis and demonstrate its effectiveness on data from actual software executions. We first describe count sketch without any privacy-related randomization. The next subsection shows how randomization can be applied to achieve the differential privacy guarantee.

Counts sketch in our setting is based on s pairs of independent hash functions (h_k, g_k), for $1 \leq k \leq s$, such that $h_k : \mathcal{T} \to \{1, \ldots, m\}$ and $g_k : \mathcal{T} \to \{+1, -1\}$. Here parameters s and m are chosen ahead of time; this choice will be discussed later. To perform analysis without differential privacy, each user would create a *local sketch* and then send it to the analysis server, where a *global sketch* is constructed and used to produce frequency estimates. The local sketch for user u_i is a $s \times m$ matrix S_i initialized with 0 elements. For every locally-covered trace $t \in T_i$, and for every $1 \leq k \leq s$, matrix element $S_i[k, h_k(t)]$ is updated by adding to it the value of $g_k(t)$. In essence, for every row k in the matrix, we use hash function h_k to hash t into a value from $\{1, \ldots, m\}$, and then update a counter for that value with $+1$ or -1 depending on hash function g_k. The local sketches S_i for all users are then sent to the analysis server, where a global sketch S_g is constructed by element-wise addition of all S_i. Finally, for any $t \in \mathcal{T}$, a frequency estimate can be obtained by reporting the median value of $S_g[k, h_k(t)] \times g_k(t)$ over all $1 \leq k \leq s$.

Example

Figure 1 illustrates the local sketch for one user, based on data obtained from our implementation on one of our benchmarks (Android app `drumpads`). We use integer method ids to denote app methods. For example, id 473 corresponds to method `MainActivity.initOnboarding` and id 971 corresponds to `OnboardingView.createImageScene`. For brevity, the example uses the method id to signify an `enter` event for the corresponding method; id 0 corresponds to the start event.

Suppose that the locally-covered chains for some user are $t_1, \ldots, t_7$. We illustrate count sketch with $m = 8$ and $s = 3$. Thus, each chain t is hashed into a value $h_k(t) \in \{1, \ldots, m\}$ using three different hash functions (i.e., $1 \leq k \leq 3$). An additional hash $g_k(t)$ produces a $+1/-1$ value. Accumulating these values, as described above, results in the local sketch shown at the bottom of the figure. For example, the first cell in the second row has a value of -2 because $h_2(t_3) = h_2(t_6) = 1$ (i.e., both chains map to this cell), and $g_2(t_3) = g_2(t_6) = -1$ (i.e, both contribute -1 to the value of the cell). This also illustrates that hashing does produce collisions. Using s pairs of hash functions helps ameliorate this problem.

In this particular example the sketch accurately preserves the original information. Consider, for example, chain t_3. The cells for this chain, as determined by hashes h_k, are $[1,5]$, $[2,1]$, and $[3,4]$ in [row,column] format. The corresponding cell values are -1, -2, and -1. The median value of $-1 \times g_1(t_3)$, $-2 \times g_2(t_3)$, and $-1 \times g_3(t_3)$ is 1, which accurately reflects the raw local data.

The advantage of using this approach is that a local sketch S_i for user u_i provides a fixed-sized representation of the arbitrary subset T_i of the set $\mathcal{T}$ of possible traces. Further, randomization of the local sketch, as described shortly, can be performed in time proportional to this $s \times m$ sketch size. Thus, instead of recording the raw data T_i and randomizing it with randomized response to achieve the differential privacy guarantee over $\mathcal{T}$, we will record the sketch S_i and randomize it to achieve the differential privacy guarantee over local sketches. Finally, the count sketch technique is theoretically proven to produce accurate estimates for high-frequency items, which aligns well with our goal to produce information about frequently-occurring traces, as discussed further in Section 4.

3.2 Sketch Randomization

Trace-level randomization

To introduce privacy-achieving randomization, for each locally-covered trace $t \in T_i$ the following actions are performed. First, for each row k in the local sketch S_i, the contribution of t to this row is expressed as a vector of length m (which is the number of columns in the sketch). The vector has the value of $g_k(t) \in \{+1, -1\}$ in position $h_k(t)$, and 0 values in all other positions. Then, the following randomization is applied to this vector:

- for each position with a 0 value, independently of any other positions in the vector, with equal probability the 0 value is replaced by $+1$ or -1
- for the position with the single $-1/+1$ value, the sign of this value is inverted with probability $1/(e^\epsilon + 1)$

The resulting randomized vector contains only $+1$ and -1 values. We can think of this process as applying a randomizer $R_k : \mathcal{T} \to \{+1, -1\}^m$. It can be proven that this approach achieves indistinguishability between t and any $t' \in \mathcal{T}$. The outline of this proof is as follows. First, consider the case when t and t' are hashed to the same position in count sketch row – that is, $h_k(t) = h_k(t')$. For any $Z \in \{+1, -1\}^m$, it is easy to see that the ratio of $P[R_k(t) = Z]$ and $P[R_k(t') = Z]$ can be bounded by the ratio of $e^\epsilon/(e^\epsilon + 1)$ (i.e., the probability that the sign at the non-zero position is preserved) and $1/(e^\epsilon + 1)$ (i.e., the probability that the sign at the non-zero position is inverted). The second case is when t and t' are hashed to different positions. Then the ratio of $P[R_k(t) = Z]$ and $P[R_k(t') = Z]$ is bounded by the ratio of $\frac{1}{2}e^\epsilon/(e^\epsilon + 1)$ and $\frac{1}{2}/(e^\epsilon + 1)$; here $\frac{1}{2}$ is the probability associated with the randomization of the zero positions. In either case, for any vector Z containing m values $+1/-1$, the probabilities $P[R_k(t) = Z]$ and $P[R_k(t') = Z]$ differ by at most a factor of e^ϵ. By observing Z, a malicious observer cannot conclude with high confidence that the underlying trace was t as opposed to any other $t' \in \mathcal{T}$.

Set-level randomization

Next we define the complete randomizer: given the local set of traces T_i, $R_k(T_i) = \sum_{t \in T_i} R_k(t)$ where the addition is element-wise. This definition satisfies the indistinguishability property in the following sense. Consider any $t \in T_i$ and any $t' \in \mathcal{T} \setminus T_i$. Let $T_i' = (T_i \setminus \{t\}) \cup \{t'\}$. Thus, T_i' is obtained by replacing t with t'. For any output Z of R_k, the probabilities $P[R_k(T_i) = Z]$ and $P[R_k(T_i') = Z]$ differ by at most a factor of e^ϵ. Thus, an observer of

Z cannot determine with high confidence that a particular trace t was present in T_i, as opposed to any other trace $t' \notin T_i$. The complete randomized local sketch is constructed as a $s \times m$ matrix in which row k is $R_k(T_i)$; we will denote this matrix by $R(S_i)$ where S_i is the non-randomized local sketch. This randomized local sketch is reported to the analysis server.

3.3 Efficient Randomization

The approach described above is impractically expensive. Specifically, for any $t \in T_i$ we need to compute s randomized vectors of length m, where each vector element requires drawing a random value. In our experience the cost of such processing could be high for data from actual software executions. Instead, we use an approach that first records the contributions of each t without randomization, and then draws random values from the binomial distribution to implement "one-shot" randomization.

Algorithm 1 describes the details of this approach. Consider a cell $[k, j]$ in the sketch. Let $N_{+1}[k, j]$ be the number of traces that contribute $+1$ to the value in this cell, without randomization. Similarly, let $N_{-1}[k, j]$ be the number of traces that contribute -1 to the cell. Our approach first records these counts (function `add`) without randomization. After data collection is completed, `finalize` computes the randomized sketch. With randomization, each of the $N_{+1}[k, j]$ occurrences of $+1$ contributes $+1$ with probability p and -1 with probability $1 - p$. Binomial distribution gives us the probability of y successes in x independent trials, where each trial succeeds with probability p. Let $binomial(x, p)$ denote a random value drawn from this distribution. The randomization will contribute $binomial(N_{+1}[k, j], p)$ values of $+1$ to the cell value; the remaining $N_{+1}[k, j] - binomial(N_{+1}[k, j], p)$ contributions will be -1. Thus, at line 19 of the algorithm we compute the cumulative contribution of the "raw" $+1$ values as the difference between these two quantities – that is, as $2 \times binomial(N_{+1}[k, j], p) - N_{+1}[k, j]$. A similar computation is performed at line 20 for the -1 values. Finally, we also have to account for the randomization of 0 values, which is done at line 21. The combined effect of these three cases is computed at line 22 as the cell value in the randomized sketch. This approach has cost in the order of $s \times m$, while a naive implementation with separate randomization for each observed trace will have cost in the order of $|T_i| \times s \times m$.

3.4 Server-Side Processing

The randomized local sketches $R(S_i)$ from all users are collected by the analysis server and their element-wise sum is computed. To obtain unbiased estimates, all elements of the sum need to be scaled by $(e^\epsilon + 1)/(e^\epsilon - 1)$. The resulting $s \times m$ matrix S_g is the *global sketch* produced by the analysis. For any $t \in \mathcal{T}$, an estimate $\hat{f}(t)$ of the true frequency $f(t)$ can be obtained as the median value of $S_g[k, h_k(t)] \times g_k(t)$ over all sketch rows k. This processing is described in Algorithm 2. It is important to note that summing up of the local sketches is essential in order for the randomized noises to "cancel out" across the population of users.

3.5 Selecting Sketch Size

The selection of sketch size is important for achieving high accuracy of estimates. In our implementation, both the number of rows s and the number of columns m are powers of 2. Parameter s is set to 256, which is similar to values used in prior work [8]. When selecting the number m of sketch columns, we aim to use a value that would produce a small number of hash collisions. One simple choice is to select m to be similar to the total number of unique traces that would be represented in the global sketch – that is, similar to the size

Algorithm 1 Randomized count sketch.

output : S_i: randomized local sketch

Function `init()`:
- $S_i \leftarrow \{0\}^{s \times m}$
- $N_{+1} \leftarrow \{0\}^{s \times m}$
- $N_{-1} \leftarrow \{0\}^{s \times m}$

Function `add(`t`)`:
- $T_i \leftarrow T_i \cup \{t\}$
- **for** $k \leftarrow 1$ **to** s **do**
  - **if** $g_k(t) = +1$ **then**
    - $N_{+1}[k, h_k(t)] \leftarrow N_{+1}[k, h_k(t)] + 1$
  - **else**
    - $N_{-1}[k, h_k(t)] \leftarrow N_{-1}[k, h_k(t)] + 1$
  - **end**
- **end**

Function `finalize()`:
- $p \leftarrow \frac{e^\epsilon}{1+e^\epsilon}$
- **for** $k \leftarrow 1$ **to** s **do**
  - **for** $j \leftarrow 1$ **to** m **do**
    - $z \leftarrow |T_i| - N_{+1}[k, j] - N_{-1}[k, j]$
    - $n_{+1} \leftarrow 2 \times \mathit{binomial}(N_{+1}[k, j], p) - N_{+1}[k, j]$
    - $n_{-1} \leftarrow 2 \times \mathit{binomial}(N_{-1}[k, j], p) - N_{-1}[k, j]$
    - $n_0 \leftarrow 2 \times \mathit{binomial}(z, \frac{1}{2}) - z$
    - $S_i[k, j] \leftarrow n_{+1} - n_{-1} + n_0$
  - **end**
- **end**

of the union of all local sets T_i. The value of m has to be selected ahead of time, before deployment, so that the randomization machinery is included in the distributed code. To make this selection, we use an approach similar in spirit to existing techniques [6, 57]. First, a group of *opt-in* users is used to obtain detailed information in a non-differentially-private manner. Specifically, the set of local traces T_i from each opt-in user u_i is collected and reported to the analysis server. Then, the union of these sets is determined. The value of m is defined as the smallest power of 2 greater than or equal to the size of this union. This value of m is then used by the *regular* software users, whose copy of the software embeds this m value and only reports the randomized sketch of their local information.

In practice, there are several options for obtaining this opt-in data. First, some users may be willing to share their raw data. Even in this case, instead of the raw data the approach could collect some hashed version of it, which provides some degree of privacy protection (although weaker than differential privacy). Alternatively, such data could also be provided from in-house testing or beta testing. In our experiments, each run of the approach randomly picks 10% of the users as opt-in users, computes m based on their data, and then performs the rest of the experiment on the remaining 90% users.

The size of the sketch produced by this approach depends on the underlying volume of collected data. Suppose, for example, that there are a total of 15 thousand unique traces across all software users, which corresponds to $m = 2^{14}$. Assuming each sketch element is represented as a 2-byte integer, the total sketch size is 8MB, which is a practical amount of

Algorithm 2 Server-side processing.

```
1  Function global_sketch(R(S_1), ..., R(S_n)):
2      S_g ← {0}^{s×m}
3      for i ← 1 to n do
4          S_g ← S_g + R(S_i)
5      end
6      S_g ← (e^ε + 1)/(e^ε − 1) × S_g
7  Function estimate(t):
8      E ← ∅
9      for k ← 1 to s do
10         E ← E ∪ { S_g[k, h_k(t)] × g_k(t) }
11     end
12     return median(E)
```

data to transfer. However, if the number of unique traces across the population of software users is many hundreds of thousands, sketch size becomes impractical. If the goal is to achieve high accuracy of estimates while having a reasonably small amount of data communication with the analysis server, our approach would be most suitable for scenarios where the total number of unique traces reported from the user population is in the order of a few thousands to a few tens of thousands. Depending on the intended use of the analysis information, this could be a reasonable constraint. For example, if the analysis data is used to identify common user behaviors for the purposes of manual performance optimization or user interface redesign, it is unlikely that frequency estimates for hundreds of thousands of traces would be of value to software developers. To achieve such data sizes, a simple approach is to use pre-defined limits on the sizes of local sets or the lengths of collected traces. Our implementation limits the length of collected call chains to 10 events and the length of collected enter/exit traces to 20 events. This also bounds the depth of exploration for hot traces, which is described next.

4 Identification of Hot Traces

From the global sketch, the analysis server can estimate the frequency of any particular trace $t \in \mathcal{T}$. However, this is not enough for many forms of data analyses, since the size of $\mathcal{T}$ is very large (or even infinite) and obtaining an estimate for each t is not possible. Next we focus on one particular data analysis of significant practical importance: identifying the *hot traces* and estimating their frequencies. Hot traces are useful in identifying common user behavior, which themselves can be used for performance optimization, focused testing and static checking, and application-flow optimization. We consider a trace t to be hot if its true frequency $f(t) \geq h$, where $h = \alpha \times n$ is a "hotness threshold" defined by a parameter α and the number of software users n. The question is, *given the global sketch, how can we efficiently and accurately construct an estimate of the set of hot traces?* Next, we develop an approach to answer this question.

Exploration of estimated hot traces

Our approach takes as input the global sketch S_g, together with the set $\mathcal{E}$ of possible run-time events, the start event $s \in \mathcal{E}$, and the family of extension functions ext. We perform a pruned exploration of the elements of $\mathcal{T}$ defined by $\mathcal{E}$ and ext. The key observation is that if a trace

t is *not* hot, any t' that has t as a prefix cannot be hot, since the number of users that covered t' cannot exceed the number of users that covered t. This leads to the following approach: starting with the length-0 trace $\langle s \rangle$, explore the space of possible trace extensions defined by ext. For each explored trace t, estimate its frequency using sketch S_g and stop the exploration if the frequency estimate $\hat{f}(t)$ is below the hotness threshold h. Otherwise, continue the exploration with all traces in $\mathsf{ext}(t)$.

A key assumption of this approach is that for any given trace t, the set of extended traces $\mathsf{ext}(t)$ can be computed efficiently. We chose the two exemplar analyses presented in Section 2 – call chains and enter/exit traces – to illustrate two common cases where this computation is naturally derived from the definition of the underlying formal language. Such trace structure is not specific to these two examples; other dynamic analyses (e.g., paths in control-flow graphs) have similar properties. For call chains, the traces are strings in a regular language. Thus, the exploration is equivalent to exploring paths in the corresponding finite-state automaton. The extension function is defined by the set of possible transitions from the current state of the automaton. For enter/exit traces, defined by a Dyck context-free language (i.e., a language of balanced parentheses), the corresponding pushdown automaton can be maintained during the exploration of strings, and the extension function is again defined by the possible transitions from the current automaton state. Our implementation of these exemplar analyses uses exactly this approach. In both cases, the transitions are efficient: the cost of computing $\mathsf{ext}(t)$ is linear in the size of this set. Note that this approach is also applicable in the more general case where $\mathcal{T}$ is defined by an arbitrary context-free grammar, as the corresponding pushdown automaton can be maintained during trace exploration and consulted to decide how to extend the current trace.

Relaxed hotness criterion

Our experience indicates that the approach described above has the following disadvantage: sometimes entire groups of hot traces with a common prefix are not discovered because this prefix is misclassified as not being hot due to an inaccuracy of its frequency estimate. As a result, the exploration stops too early. To address this problem, we designed a more robust "relaxed" check for hot traces. If for some explored trace t we have $h/2 \leq \hat{f}(t) < h$, we consider this trace a potentially-misclassified hot trace due to an inaccurate estimate. In such cases, we check whether at least one $t' \in \mathsf{ext}(t)$ has an estimate above the threshold h. If such a t' exists, we take it as strong indication that t itself is hot and treat it as such. The details of the entire approach are presented in Algorithm 3.

For illustration, consider an enter/exit trace derived from actual data for the `equibase` app, which is one of our experimental subjects. The trace is $t = \langle \mathsf{enter}(0), \mathsf{enter}(1685), \mathsf{enter}(1678), \mathsf{enter}(910), \mathsf{enter}(805), \mathsf{enter}(10), \mathsf{exit}(10), \mathsf{exit}(805), \mathsf{exit}(910), \mathsf{enter}(1677) \rangle$. The true frequency is $f(t) = 818$. For the hotness cut-off $h = 810$ which was used in that experiment, the trace is hot. However, because of estimate $\hat{f}(t) = 763$, the exploration will stop at this trace if the relaxed criterion is not employed. As a result, 15 hot traces that have t as a prefix would be missed. Using the relaxed criterion, all 15 traces are correctly discovered by Algorithm 3.

5 Evaluation

For evaluation, we used 15 Android applications that were used by prior related work [59, 58]. We simulated 1000 users interacting with each app using the Monkey tool [23]. Specifically, we performed 1000 independent Monkey runs and considered each Monkey execution as triggered by one simulated user. During this process, for each run, we collected the sequence

Algorithm 3 Identification of hot traces.

```
   output : H: set of estimated hot traces
 1 Function hot_traces():
 2 |   H ← ∅
 3 |   for t ∈ ext(s) do explore(t)
 4 Function explore(t):
 5 |   if hot(t) then
 6 |   |   H ← H ∪ {t}
 7 |   |   for t′ ∈ ext(t) do explore(t′)
 8 Function hot(t):
 9 |   e ← f̂(t)
10 |   if e ≥ h then return true
11 |   if e < h/2 then return false
12 |   for t′ ∈ ext(s) do
13 |   |   if f̂(t′) ≥ h then return true
14 |   end
15 |   return false
```

of method enter/exit events until the total number of enter events reaches 10× the number of methods defined in the static app code (excluding libraries). If the app crashed or Monkey triggered events very slowly, we restarted Monkey and continued collecting the trace for this simulated user until the total number of enter events reached this targeted value. From this sequence of enter/exit events we constructed the set of observed call chains for that simulated user u_i – that is, set T_i for call chain analysis. In addition, we also considered the entry methods of the app and collect the subsequences that start at the enter events of those methods; these subsequences form set T_i for enter/exit trace analysis. Thus, for each of the two analyses we gathered sets $T_1, T_2, \ldots, T_{1000}$. We also wanted to study the effects of the number of users, but since execution of a large number of Monkey runs in device emulators takes a very long time, we employed an approach used by others [59]: each of the 1000 sets was replicated 10 times to generate T_i for $n = 10000$.

Our trace collection approach creates a threat to validity: it is well known that the app coverage achieved via tools such as Monkey can be limited [11]. In general, data generated by automated GUI crawling may not be representative of the behavior of real-world app users. One indication of coverage for our experiments is the size of $\cup_i T_i$, shown in columns "Total" in Table 1. For most apps, more than a thousand different traces were observed.

The instrumentation is based on the Soot code rewriting tool [47]. Only application code is instrumented, as this is the most likely focus of interest for app developers. We treat the following methods as app entry methods: methods that implement/override any Android framework methods (e.g., callbacks such as `onClick`); `<clinit>` methods; and `<init>` methods of application subclasses of Android framework classes.

Given the data collected by the instrumentation, we ran all randomization separately from the executions that gather the traces. This allowed us to conduct each experiment 30 times, in order to report rigorous statistical results that account for the randomness introduced by local randomizers [20]. Experiments were performed for several values of ϵ used in prior work [18, 52, 59, 58]. For brevity, most results are presented for $\epsilon = \ln(9)$, but the effects of other values are also discussed. To implement count sketch, we used SHA-256

Table 1 Experimental subjects and analyzed traces.

App	Classes	Call Chains				Enter/Exit Traces			
		Total	Len_{avg}	Time_u	Time_s	Total	Len_{avg}	Time_u	Time_s
`barometer`	379	2765	4.3	0.3	25	2717	10.2	0.4	6.4
`bible`	1107	1604	3.3	0.2	64	2427	8.8	0.2	21
`dpm`	272	1272	4.0	0.1	4.3	2475	10.6	0.2	3.7
`drumpads`	447	926	3.4	0.1	6	1289	8.8	0.1	4.1
`equibase`	252	773	3.0	0.1	3.2	1602	9.1	0.3	4.9
`localtv`	716	4037	4.6	0.3	42	5285	10.4	0.3	12
`loctracker`	198	480	1.8	0.1	0.8	1098	6.2	0.1	8.9
`mitula`	973	24757	7.0	2.8	1784	5614	10.2	0.8	27
`moonphases`	166	1755	6.4	0.2	3.3	947	9.9	0.1	0.6
`parking`	379	1477	3.1	0.1	10	2875	8.8	0.2	4.6
`parrot`	1099	7575	4.7	0.8	427	6499	10.0	0.9	63
`post`	1107	2358	3.8	0.4	92	3564	9.9	0.5	45
`quicknews`	1107	3668	3.4	0.4	51	6062	8.9	0.7	57
`speedlogic`	86	244	3.0	0.0	0.1	304	8.1	0.0	0.3
`vidanta`	1608	7811	4.9	0.8	833	6687	9.7	0.9	124

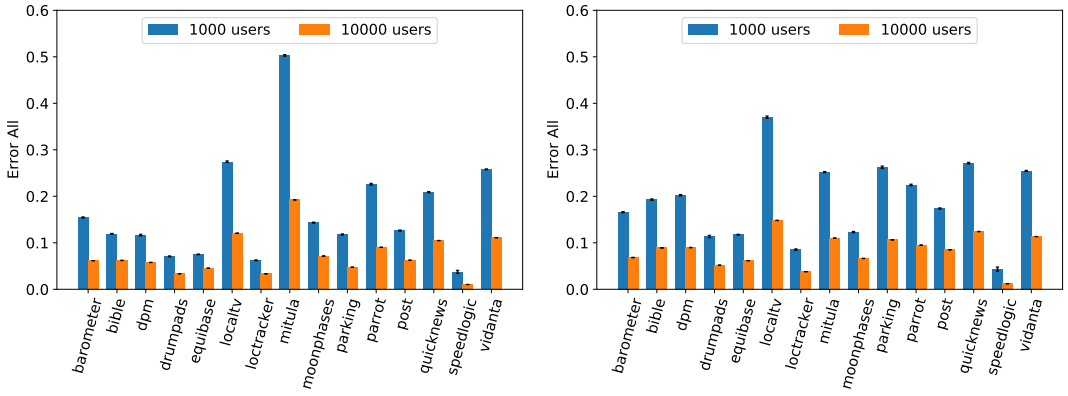
hashing. In particular, hash functions h_k and g_k used in count sketch were implemented by prepending k to the string representation of the trace (which itself is based on the methods ids), computing SHA-256, and taking the appropriate number of bits from the result.

Table 1 shows the details of the subjects used in our experiments. Column "Classes" lists the number of application classes, excluding several well-known third-party Android libraries, e.g., `dagger` and `okio`. The group of columns labeled "Call Chains" describes measurements for the call chain analysis, and the group labeled "Enter/Exit Traces" shows the same measurements for the analysis of enter/exit traces. Column "Total" and "Len_{avg}" show the total number of unique traces across the 1000 local sets T_i and their average length respectively. Column "Time_u" shows the average time (in seconds) to process the local data of a user, as described in Algorithm 1. Column "Time_s" contains the time (in seconds) to identify hot traces from the global sketch at the analysis server, using the approach from Algorithm 2 for $n = 1000$. For both analyses, the costs are practical and suitable for real-world use.

As mentioned in Section 3.3, we initially attempted to perform randomization separately for each covered trace, but incurred high running times for the local randomizer. This led to the development of the optimized approach in Algorithm 1. For example, for the `parrot` app, the naive randomization of call chains and enter/exit traces took 264 seconds per user on average, while the optimized one took 1.7 seconds. We typically observed two orders of magnitude improvement in the running time of the local randomizer.

5.1 Accuracy for All Covered Traces

The first research question we consider is this: *What is the accuracy of estimates for traces that are covered by at least one user?* Note that, from the data in the global sketch, the analytics server cannot directly determine this set of traces. (We address this issue in the next subsection.) However, the knowledge of this accuracy provides a useful baseline. To answer this question, we use a normalized L_1 distance between the vector of true frequencies



(a) Call chains. **(b)** Enter/exit traces.

Figure 2 Error of estimates for all covered traces.

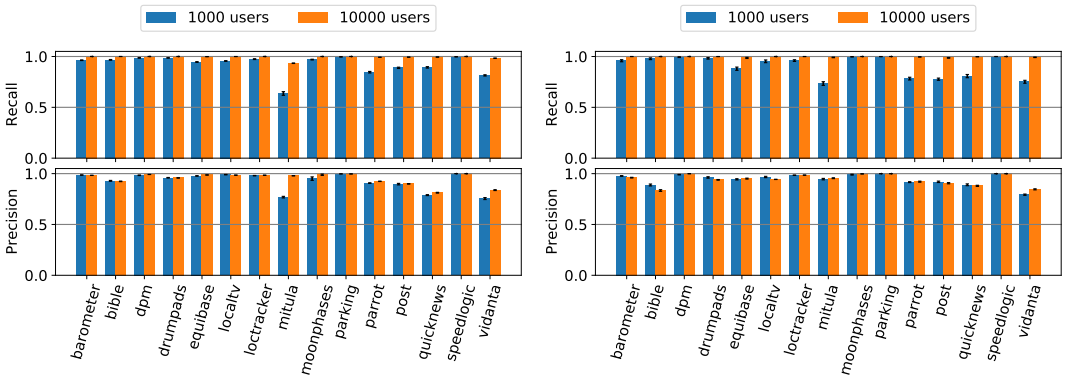
and the vector of their estimates. Specifically, for all t that appear in at least one T_i, we compute the error as $\sum_t |f(t) - \hat{f}(t)| / \sum_t f(t)$. Values close to 0 mean that the estimates are overall close to the real frequencies. Figure 2 shows these measurements for the two values of n. As described in Section 3.5, each run of this experiment (and all later experiments) used a randomly-selected set of size $n/10$ as opt-in users, and then performed the analysis and computed all reported error measurements for the remaining users. For these and all other experiments reported later, we followed a popular approach for statistically-rigorous performance measurements [20]: 30 independent runs of the experiment were performed, and the mean together with the 95% confidence interval are reported. The confidence interval characterizes the variance due to the randomization. In the bar charts, the confidence interval is shown at the top of the corresponding bar. In many cases, the interval is so small (i.e., the variance is so low), that it is hard to see in the figures.

From this data, we reach the following answer to the above question: with sufficiently large number of users participating in the data collection, the estimates are close to the real frequencies. For example, for the call chain analysis with $n = 10000$, the cumulative error over all t is under 20% in all cases, and its average value across the 15 apps is 7.4%. Similarly, for the enter/exit trace analysis with $n = 10000$, the cumulative error over all covered traces is always under 15% and, averaged across the apps, is 8.4%. It is fairly common for Android apps to have many thousands of users, and popular apps usually have hundreds of thousands of users. Thus, obtaining data from a sufficient number of app users should be feasible.

5.2 Precision and Recall for Hot Traces

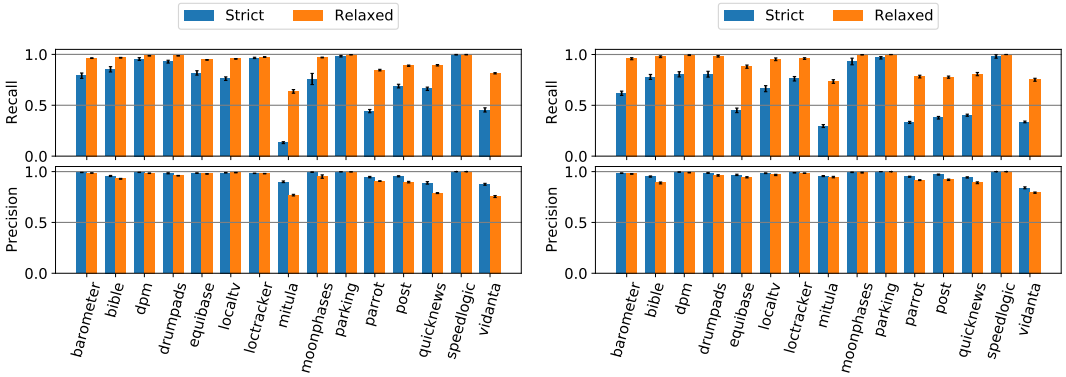
As discussed earlier, the set of all covered traces is not directly known to the analysis server. Section 4 discussed an approach to identify *hot traces*. Our next research question is: *How accurately are the hot traces identified?* The metrics we use to answer this question are recall (what portion of the true hot traces are discovered) and precision (what portion of the reported hot traces are actually hot). We executed Algorithm 3 on the global sketch to identify likely hot traces, with hotness threshold $h = 0.9 \times n$. This was done in 30 independent repetitions of the experiment. The mean values from these experiments and their 95% confidence intervals are shown in Figure 3.

Overall, the results of this experiment provide strong indication that hot traces can indeed be identified accurately with a sufficient number of users. For $n = 1000$, the average recall across the 15 apps is 92.1% and the average precision is 92.5% for call chains, and 90.4% and



(a) Call chains. **(b)** Entry/exit traces.

Figure 3 Recall and precision for hot traces.



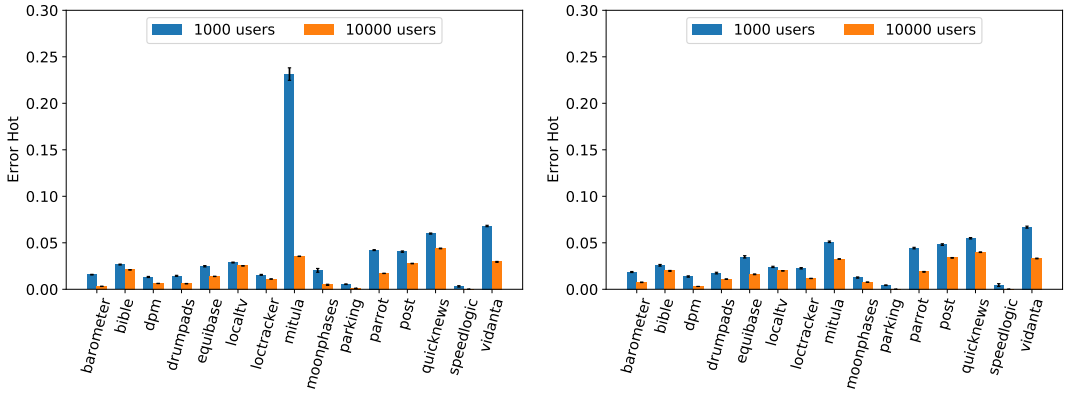
(a) Call chains. **(b)** Enter/exit traces.

Figure 4 Recall and precision for hot traces: strict vs relaxed hotness criterion.

94.5% for enter/exit traces respectively. For $n = 10000$, the recall for call chains increases to 99.3% and the precision to 95.0%; for enter/exit traces, the recall increases to 99.7% and precision decreases slightly to 94.1%. We investigated the apps with the lowest precision and determined that they have a large number of traces whose true frequencies are slightly below the threshold h; some of these almost-hot traces are misclassified as being hot, leading to the lower precision.

One related question is how the design choices for Algorithm 3 affect its precision and recall. In Section 4, we discussed two possible criteria for deciding whether a trace should be considered hot. The "strict" criterion is that a trace's estimate $\hat{f}(t)$ should exceed the hotness threshold h. However, if this estimate is inaccurate and too small, the chain and all other hot chains that have it as prefix will be missed. Thus, in the algorithm we use a "relaxed" criterion which also considers traces t with estimates $h/2 \leq \hat{f}(t) < h$ such that t has at least one extended trace with an estimate that exceeds h. This relaxed criterion was employed when collecting the data in Figure 3.

To understand the effects of this choice, we also measured precision and recall using the strict criterion. Figure 4 shows a comparison between the two criteria for $n = 1000$; the other value for n leads to similar conclusions. As can be seen from these measurements, using the strict criterion results in lower recall. For example, for call chain analysis, three apps have



(a) Call chains. **(b)** Enter/exit traces.

Figure 5 Error of estimates for reported hot traces.

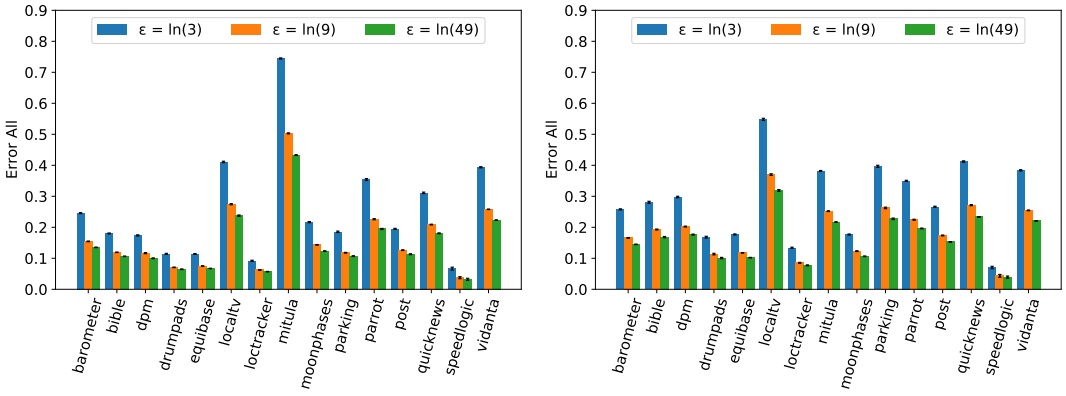
recall less than 50%. Similarly, for enter/exit trace analysis there are six apps with recall below 50%. As expected, the strict criterion does improve precision, but this effect is not very pronounced. Depending on the intended uses of the analysis, the app developers may prefer higher recall or higher precision. Using these two criteria, or variations of them, allows this trade-off to be adjusted as desired.

5.3 Accuracy of Estimates for Reported Hot Traces

For the set of traces reported by Algorithm 3 as likely-hot, we ask following question: *What is the accuracy of estimates for reported hot traces?* Figure 5 shows the error of estimates, using a metric similar to the one used in Figure 2: the sum of $|\hat{f}(t) - f(t)|$ for all reported hot traces t, normalized by the sum of $f(t)$ for those t. Based on these results, the answer to the question is that high accuracy is achieved for the frequency estimates of hot traces. Combined with the high recall demonstrated earlier, our conclusion is that hot traces and their frequencies can be successfully estimated via our differentially-private analysis.

Compared to other apps, in Figure 5a the error for app `mitula` is significantly larger for 1000 users. The underlying reasons are indicated in Figure 2a, where the estimates for this app have large cumulative error for 1000 users. This produces a large number of false positive hot chains (Figure 3a); further, those false positives have significant cumulative error. If we remove the false-positive hot chains from Figure 5a, the cumulative error becomes similar to that for the other apps. The reason for the error in Figure 2a is that there are many more chains in this app compared to the other apps. Further, the distribution of the frequency of these chains is not uniform: there is a large number of low-frequency chains, and the DP approaches produce inaccurate estimates for such chains. This can be solved by increasing the number of users (as can be seen in all figures for 10000 users): even the low-frequency chains now have enough instances to benefit from "random noise cancellation" across a larger number of instances.

It is instructive to compare Figure 5 with Figure 2. Overall, the estimate error for hot traces is smaller than the estimate error for all traces. For example, for $n = 10000$, the average error value in Figure 5a is 1.6%, compared to 7.4% in Figure 2a, and 1.7% vs 8.4% for Figure 5b vs Figure 2b. Theoretically, both count sketch and randomized response tend to favor higher-frequency items. The higher accuracy for hot traces demonstrates that this also holds in practice.



(a) Call chains. **(b)** Enter/exit traces.

Figure 6 Error of estimates for all covered traces for three values of ϵ.

5.4 Privacy Loss Parameter

As discussed earlier, the privacy loss parameter ϵ can be used to tune accuracy/privacy trade-offs. We considered the following question: *To what degree does accuracy change with changes in this parameter?* In existing work, ϵ ranges from 0.01 to 10 [28]. Related work that employs randomized response has used, for example, ln(3), ln(9), and ln(49) [18, 59, 58]. We computed the error for all covered traces for these three values; the results for ln(9) were already presented in Figure 2 and are repeated here. Figure 6 shows these measurements for $n = 1000$; similar trends are seen for the other n value. Overall, with increasing ϵ, the expected accuracy gains are observed but seem to level off. For call chains and enter/exit traces, respectively, the average error across all apps decreases from 25.3% and 28.7% for the smallest value of ϵ to 16.6% and 19.0% for ln(9), and then further to 14.5% and 16.5% for the largest value of the parameter. Based on these results, we consider ln(9) to provide a reasonable trade-off and have used it to present the majority of data in our evaluation. In practical scenarios, the developers can select a small fixed value of ϵ (before deployment), based on data from in-house testing or from real opt-in users, as well as the desired accuracy. Once selected, ϵ provides an upper bound on the privacy loss: for any data, and any two data items, they are guaranteed to be ϵ-indistinguishable. The real workload will affect only the accuracy, not the privacy.

5.5 Summary of Results

Our experimental results can be summarized as follows. First, as illustrated in Figure 2, the frequency estimates have high accuracy, for practical values of ϵ. This results indicates that with good privacy and sufficient number of software users, one can obtain accurate frequency estimates for software traces. Second, based on the results in Figure 3, the set of hot traces can be determined with high precision and recall. The relaxed identification of hot traces is important for achieving this result (Figure 4). Third, the frequency estimates for hot traces are accurate and better than those for the remaining covered traces (Figure 5). Finally, consider the accuracy/privacy trade-off spectrum: from smaller values of ϵ (i.e., stronger privacy) and lower accuracy, to larger values of ϵ and high accuracy. As indicated by Figure 6, after a certain point in this spectrum there do not seem to be significant additional improvements in accuracy.

6 Related Work

Differential privacy

There is a large body of work on both the theory and practice of differential privacy. As already discussed, several approaches based on randomized response consider a single data item per user [18, 52, 8], while we are interested in a set of data items (i.e., a set of locally-covered traces). Differentially-private analysis of software executions has also been studied in prior work [58, 57, 60]. In those efforts the domain of possible items is small, enumerated ahead of time before software deployment, and the randomizer output is straightforward to generate and store. A key distinguishing feature of our work is that the domain of possible traces is either infinite or very large, which requires different randomization techniques. We address this problem by using a count sketch representation. This allows tunable trade-offs between accuracy and representation size, as well as higher accuracy for high-frequency traces. Efficient randomization of simple bitvectors has also been considered [58]. Our efficient randomization (Section 3.3) requires more general reasoning. Because of the small number of possible data items, these prior efforts do not need to explore a large domain in order to identify hot items. In contrast, we need to develop effective search in a domain containing billions of possible traces. We demonstrate how to achieve this using considerations of chain prefixes and suffixes, and illustrate the approach for context-free-language domains by exploring the states of the corresponding automaton (Section 4).

Privacy-preserving techniques in programming languages and software engineering

The programming languages community has investigated techniques for testing and verification of differentially-private algorithms and implementations [56, 53, 36, 61]. Privacy issues are also important for many areas in software engineering, including design [25], testing [24, 9, 50, 32], and defect prediction [44, 45, 31]. Other than the work described earlier, we are not aware of attempts to employ differential privacy techniques in this area. Given the strong theoretical properties of such techniques, and their increasing adoption in industry and government [33, 4, 18, 51, 14], it is a worthwhile research goal to reconsider a range of software engineering techniques using differential privacy machinery.

Analysis of deployed software

Remote analysis of deployed software is an area with a significant body of prior work. As one example, residual coverage monitoring [43] uses coverage information from software users for testing purposes. GAMMA [42] collects data from software users and orchestrates the data collection across program instances. Placement of profiling probes has been considered by several projects [15, 39]. Failure reproduction and debugging are aided by collected data from deployed software [12]. Similarly, researchers have proposed analysis of post-deployment failure reports [38].

Privacy in remote software analysis has been targeted by prior efforts. Anonymization of collected data has taken several forms [17, 13]. As shown by privacy researchers [34, 35], anonymization is not enough to provide strong privacy guarantees. Instead, we consider the principled protection provided by local differential privacy. Remote software analyses from prior work could potentially benefit from developing differentially-private versions for them. Examples of such analyses include impact analysis and regression testing [41], as well as failure analysis [27, 29, 30].

7 Conclusions and Future Work

Differential privacy is a promising approach for developing new privacy-preserving software analyses. The growing adoption of differential privacy for practical use, together with its rigorous foundations, provide further motivation to study such analyses. We develop the design of a differentially-private trace coverage analysis, based on an incremental definition of the trace domain. We employ local count sketches, randomize them efficiently, and analyze them at the server side to obtain frequency estimates and to search for hot traces. The approach is illustrated with a call chain analysis and an enter/exit trace analysis. Our experimental studies present promising findings: with realistic numbers of software users, one can use these privacy-preserving techniques to obtain accurate frequency estimates for trace coverage and to effectively identify hot traces.

There is a large body of prior work on software analysis that could be revisited with increased emphasis on privacy in general, and differential privacy in particular [42, 41, 27, 12, 13, 29, 30, 38]. Such studies will contribute to broader efforts to integrate privacy-preserving techniques in the analysis of deployed software, in response to growing needs for better privacy of data collection.

References

1 ACM SIGACT/EATCS. Gödel Prize. `https://sigact.org/prizes/g%C3%B6del/citation2017.pdf`, 2017.

2 L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

3 G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI*, page 85–96, 1997.

4 Apple. Learning with privacy at scale. `https://machinelearning.apple.com/2017/12/06/learning-with-privacy-at-scale.html`, 2017.

5 M. Arnold, S.J. Fink, D. Grove, M. Hind, and P.F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

6 B. Avent, A. Korolova, D. Zeber, T. Hovden, and B. Livshits. BLENDER: Enabling local search with a hybrid differential privacy model. In *USENIX Security*, pages 747–764, 2017.

7 T. Ball and J. Larus. Optimally profiling and tracing programs. *TOPLAS*, 16(4):1319–1360, 1994.

8 R. Bassily, K. Nissim, U. Stemmer, and A. Thakurta. Practical locally private heavy hitters. In *NIPS*, pages 2285–2293, 2017.

9 A. Budi, D. Lo, L. Jiang, and Lucia. kb-anonymity: A model for anonymized behaviour-preserving test and debugging data. In *PLDI*, pages 447–457, 2011.

10 M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *ICALP*, pages 693–703, 2002.

11 S.R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? In *ASE*, pages 429–440. IEEE, 2015.

12 J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *ICSE*, pages 261–270, 2007.

13 J. Clause and A. Orso. Camouflage: Automated anonymization of field data. In *ICSE*, pages 21–30, 2011.

14 A. Dajan, A. Lauger, P. Singer, D. Kifer, J. Reiter, A. Machanavajjhala, S. Garfinkel, S. Dahl, M. Graham, V. Karwa, H. Kim, P. Leclerc, I. Schmutte, W. Sexton, L. Vilhuber, and J. Abowd. The modernization of statistical disclosure limitation at the U.S. Census Bureau. `https://www2.census.gov/cac/sac/meetings/2017-09/statistical-disclosure-limitation.pdf`, 2017.

15 M. Diep, M. Cohen, and S. Elbaum. Probe distribution techniques to profile events in deployed software. In *ISSRE*, pages 331–342, 2006.

16 C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

17 S. Elbaum and M. Hardojo. An empirical study of profiling strategies for released software and their impact on testing activities. In *ISSTA*, pages 65–75, 2004.

18 Ú. Erlingsson, V. Pihur, and A. Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *CCS*, pages 1054–1067, 2014.

19 Facebook. Facebook analytics. `https://analytics.facebook.com`, 2020.

20 A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA*, page 57–76, 2007.

21 Google. Google Analytics. `https://analytics.google.com`.

22 Google. Firebase Analytics. `https://firebase.google.com`, 2020.

23 Google. Monkey: UI/Application exerciser for Android. `https://developer.android.com/studio/test/monkey`, 2020.

24 M. Grechanik, C. Csallner, C. Fu, and Q. Xie. Is data privacy always good for software testing? In *ISSRE*, pages 368–377, 2010.

25 I. Hadar, T. Hasson, O. Ayalon, E. Toch, M. Birnhack, S. Sherman, and A. Balissa. Privacy by designers: Software developers' privacy mindset. *Empirical Software Engineering*, 23(1):259–289, 2018.

26 S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE*, pages 145–155, 2012.

27 M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying classification techniques to remotely-collected program execution data. In *ESEC/FSE*, pages 146–155, 2005.

28 J. Hsu, M. Gaboardi, A. Haeberlen, S. Khanna, A. Narayan, B. C. Pierce, and A. Roth. Differential privacy: An economic method for choosing epsilon. In *CSF*, pages 398–410, 2014.

29 W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. In *ICSE*, pages 474–484, 2012.

30 W. Jin and A. Orso. F3: Fault localization for field failures. In *ISSTA*, pages 213–223, 2013.

31 Z. Li, X. Jing, X. Zhu, H. Zhang, B. Xu, and S. Ying. On the multiple sources and privacy preservation issues for heterogeneous defect prediction. *IEEE Transaction on Software Engineering*, pages 1–21, 2017.

32 Lucia, D. Lo, L. Jiang, and A. Budi. kbe-anonymity: Test data anonymization for evolving programs. In *ASE*, pages 262–265, 2012.

33 Microsoft. New differential privacy platform co-developed with Harvard's OpenDP unlocks data while safeguarding privacy. `https://blogs.microsoft.com/on-the-issues/2020/06/24/differential-privacy-harvard-opendp`, 2020.

34 A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *S&P*, pages 111–125, 2008.

35 A. Narayanan and V. Shmatikov. De-anonymizing social networks. In *S&P*, pages 173–187, 2009.

36 J. P. Near, D. Darais, C. Abuah, T. Stevens, P. Gaddamadugu, L. Wang, N. Somani, M. Zhang, N. Sharma, A. Shan, and D. Song. Duet: An expressive higher-order language and linear type system for statically enforcing differential privacy. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), 2019.

37 Oath. Flurry. `http://flurry.com`.

38 P. Ohmann, A. Brooks, L. D'Antoni, and B. Liblit. Control-flow recovery from partial failure reports. In *PLDI*, pages 390–405, 2017.

39 P. Ohmann, D. B. Brown, N. Neelakandan, J. Linderoth, and B. Liblit. Optimizing customized program coverage. In *ASE*, pages 27–38, 2016.

40 OpenDP. `https://projects.iq.harvard.edu/opendp`, 2020.

41 A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE*, pages 128–137, 2003.

42 A. Orso, D. Liang, M. J. Harrold, and R. Lipton. GAMMA system: Continuous evolution of software after deployment. In *ISSTA*, pages 65–69, 2002.

43 C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *ICSE*, pages 277–284, 1999.

44 F. Peters and T. Menzies. Privacy and utility for defect prediction: Experiments with MORPH. In *ICSE*, pages 189–199, 2012.

45 F. Peters, T. Menzies, L. Gong, and H. Zhang. Balancing privacy and utility in cross-company defect prediction. *IEEE Transaction on Software Engineering*, 39(8):1054–1068, 2013.

46 T. Reps. Program analysis via graph reachability. *IST*, 40(11-12):701–726, 1998.

47 Soot. Soot analysis framework. `https://soot-oss.github.io/soot`, 2020.

48 M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

49 W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. *IEEE Transaction on Software Engineering*, 38(5):1160–1177, 2012.

50 K. Taneja, M. Grechanik, R. Ghani, and T. Xie. Testing software in age of data privacy: A balancing act. In *ESEC/FSE*, pages 201–211, 2011.

51 Uber. Uber releases project for differential privacy. `https://medium.com/uber-security-privacy/differential-privacy-open-source-7892c82c42b6`, July 2017.

52 T. Wang, J. Blocki, N. Li, and S. Jha. Locally differentially private protocols for frequency estimation. In *USENIX Security*, pages 729–745, 2017.

53 Y. Wang, Z. Ding, G. Wang, D. Kifer, and D. Zhang. Proving differential privacy with shadow execution. In *PLDI*, pages 655–669, 2019.

54 S. Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 309(60):63–69, 1965.

55 A. Wood, M. Altman, A. Bembenek, M. Bun, M. Gaboardi, J. Honaker, K. Nissim, D. O'Brien, T. Steinke, and S. Vadhan. Differential privacy: A primer for a non-technical audience. *Vanderbilt Journal of Entertainment and Technology Law*, 21(1):209–276, 2018.

56 D. Zhang and D. Kifer. LightDP: Towards automating differential privacy proofs. In *PLDI*, pages 888–901, 2017.

57 H. Zhang, Y. Hao, S. Latif, R. Bassily, and A. Rountev. Differentially-private software frequency profiling under linear constraints. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020.

58 H. Zhang, Y. Hao, S. Latif, R. Bassily, and A. Rountev. A study of event frequency profiling with differential privacy. In *CC*, page 51–62, 2020.

59 H. Zhang, S. Latif, R. Bassily, and A. Rountev. Introducing privacy in screen event frequency analysis for Android apps. In *SCAM*, pages 268–279, 2019.

60 H. Zhang, S. Latif, R. Bassily, and A. Rountev. Differentially-private control-flow node coverage for software usage analysis. In *USENIX Security*, pages 1021–1038, 2020.

61 H. Zhang, E. Roth, A. Haeberlen, B. Pierce, and A. Roth. Testing differential privacy with dual interpreters. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), November 2020.

62 X. Zhuang, M. Serrano, H. W Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI*, pages 263–271, 2006.

Idris 2: Quantitative Type Theory in Practice

Edwin Brady
School of Computer Science, University of St Andrews, Scotland, UK

Abstract

Dependent types allow us to express precisely *what* a function is intended to do. Recent work on Quantitative Type Theory (QTT) extends dependent type systems with *linearity*, also allowing precision in expressing *when* a function can run. This is promising, because it suggests the ability to design and reason about resource usage protocols, such as we might find in distributed and concurrent programming, where the state of a communication channel changes throughout program execution. As yet, however, there has not been a full-scale programming language with which to experiment with these ideas. Idris 2 is a new version of the dependently typed language Idris, with a new core language based on QTT, supporting linear and dependent types. In this paper, we introduce Idris 2, and describe how QTT has influenced its design. We give examples of the benefits of QTT in practice including: expressing which data is erased at run time, at the type level; and, resource tracking in the type system leading to type-safe concurrent programming with session types.

2012 ACM Subject Classification Software and its engineering → Functional languages

Keywords and phrases Dependent types, linear types, concurrency

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.9

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.10`

Funding This work was funded by EPSRC grant EP/T007265/1.

Acknowledgements This work has benefitted from the many contributions to the Idris 2 project from the community. I am also grateful to the anonymous referees for their helpful feedback.

1 Introduction

Dependently typed programming languages, such as Idris [8], Agda [35], and Haskell with the appropriate extensions enabled [47], allow us to give precise types which can describe assumptions about and relationships between inputs and outputs. This is valuable for reasoning about *functional* properties, such as correctness of algorithms on collections [28], termination of parsing [14] and scope safety of programs [2]. However, reasoning about *non-functional* properties in this setting, such as memory safety, protocol correctness, or resource safety in general, is more difficult though it can be achieved with techniques such as embedded domain specific languages [9] or indexed monads [3, 27]. These are, nevertheless, heavyweight techniques which can be hard to compose.

Substructural type systems, such as linear type systems [45, 33, 6], allow us to express *when* an operation can be executed, by requiring that a linear resource be accessed *exactly once*. Being able to combine linear and dependent types would give us the ability to express an ordering on operations, as required by a protocol, with precision on exactly what operations are allowed, at what time. Historically, however, a difficulty in combining linear and dependent types has been in deciding how to treat occurrences of variables in *types*. This can be avoided [26] by never allowing types to depend on a linear term, but more recent work on Quantitative Type Theory (QTT) [4, 29] solves the problem by assigning a *quantity* to each binder, and checking terms at a specific *multiplicity*. Informally, in QTT, variables and function arguments have a multiplicity: 0, 1 or unrestricted (ω). We can freely use any variable in an argument with multiplicity 0 – e.g., in types – but we can not use a variable with multiplicity 0 in an argument with any other multiplicity. A variable bound with multiplicity 1 must be used exactly once. In this way, we can describe linear resource usage protocols, and furthermore clearly express *erasure* properties in types.

Idris 2 is a new implementation of Idris, which uses QTT as its core type theory. In this paper, we explore the possibilities of programming with a full-scale language built on QTT. By full-scale, we mean a language with high level features such as inference, interfaces, local function definitions and other syntactic sugar. As an example, we will show how to implement a library for concurrent programming with session types [21]. We choose this example because, as demonstrated by the extensive literature on the topic, correct concurrent programming is both hard to achieve, and vital to the success of modern software engineering. Our aim is to show that a language based on QTT is an ideal environment in which to implement accessible tools for software developers, based on decades of theoretical results.

1.1 Contributions

This paper is about exploring what is possible in a language based on Quantitative Type Theory (QTT), and introduces a new implementation of Idris. We make the following research contributions:

- We describe Idris 2 (Section 2), the first implementation of quantitative type theory in a full programming language, and the first language with full first-class dependent types implemented in itself.
- We show how Idris 2 supports two important applications of quantities: *erasure* (Section 3.2) which gives type-level guarantees as to which values are required at run-time, and *linearity* (Section 3.3) which gives type-level guarantees of resource usage. We also describe a general approach to implementing linear resource usage protocols (Section 4).
- We give an example of QTT in practice, encoding bidirectional session types (Section 5) for safe concurrent programming.

We do not discuss the metatheory of QTT, nor the trade-offs in its design in any detail. Instead, our interest is in discussing how it has affected the design of Idris 2, and in investigating the new kinds of programming and reasoning it enables. Where appropriate, we will discuss the intuition behind how argument multiplicities work in practice.

2 An Overview of Idris

Idris is a purely functional programming language, with *first-class* dependent types. That is, types can be computed, passed to and returned from functions, and stored in variables, just like any other value. In this section, we give a brief overview of the fundamental features which we use in this paper. A full tutorial is available online[1]. Readers who are already familiar with Idris may skip to Section 2.4 which introduces the new implementation.

2.1 Functions and Data Types

The syntax of Idris is heavily influenced by the syntax of Haskell. Function application is by juxtaposition and, like Haskell and ML and other related languages, functions are defined by recursive pattern matching equations. For example, to append two lists:

```
append : List a -> List a -> List a
append [] ys = ys
append (x :: xs) ys = x :: append xs ys
```

[1] https://idris2.readthedocs.io/en/latest/tutorial/index.html

The first line is a *type declaration*, which is required in Idris[2]. Names in the type declaration which begin with a lower case letter are *type-level variables*, therefore `append` is parameterised by an element type. Data types, like `List`, are defined in terms of their *type constructor* and *data constructors*:

```
data List : Type -> Type where
     Nil  : List elem
     (::) : elem -> List elem -> List elem
```

The type of types is `Type`. Therefore, this declaration states that `List` is parameterised by a `Type`, and can be constructed using either `Nil` (an empty list) or `::` ("cons", a list consisting of a head element and and tail). As we'll see in more detail shortly, types in Idris are first-class, thus the type of `List` (`Type -> Type`) is an ordinary function type. Syntax sugar allows us to write `[]` for `Nil`, and comma separated values in brackets expand to applications of `::`, e.g. `[1, 2]` expands to `1 :: 2 :: Nil`.

2.2 Interactive Programs

Idris is a pure language, therefore functions have no side effects. Like Haskell [37], we write interactive programs by *describing* interactions using a parameterised type `IO`. For example, we have primitives for console I/O, including:

```
putStrLn : String -> IO ()
getLine : IO String
```

`IO t` is the type of an interactive action which produces a result of type `t`. So, `getLine` is an interactive action which, when executed, produces a `String` read from the console. Idris the language *evaluates* expressions to produce a description of an interactive action as an `IO t`. It is the job of the run time system to *execute* the resulting action. Actions can be chained using the >>= operator:

```
(>>=) : IO a -> (a -> IO b) -> IO b
```

For example, to read a line from the console then echo the input:

```
echo : IO ()
echo = getLine >>= \x => putStrLn x
```

For readability, again like Haskell, Idris provides `do`-notation which translates an imperative style syntax to applications of >>=. The following definition is equivalent to the definition of `echo` above.

```
echo : IO ()
echo = do x <- getLine
          putStrLn x
```

The translation from `do`-notation to applications of >>= is purely syntactic. In practice therefore we can use `do`-notation in other contexts: for example, there is a `Monad` implementation for `IO`, and we will define an alternative >>= when implementing linear resource protocols.

[2] Note that unlike Haskell, we use a single colon for the type declaration.

2.3 First-Class Types

The main distinguishing feature of Idris compared to other languages, even some other languages with dependent types, is that types are *first-class*. For example we can pass them as arguments to functions, return them from functions, or store them in variables. This enables us to define type synonyms, to compute types from data, and express relationships between and properties of data. As an initial example, we can define a *type synonym*:

```
Point : Type
Point = (Int, Int)
```

Wherever the type checker sees `Point` it will evaluate it, and treat it as `(Int, Int)`:

```
moveRight : Point -> Point
moveRight (x, y) = (x + 1, y)
```

Languages often include type synonyms as a special feature (e.g. `typedef` in C or `type` declarations in Haskell). In Idris, no special feature is needed.

2.3.1 Computing Types

First-class types allow us to compute types from data. A well-known example is `printf` in C, where a format string determines the types of arguments to be printed. C compilers typically use extensions to check the validity of the format string; first-class types allow us to implement a `printf`-style variadic function, with compile time checking of the format. We begin by defining valid formats (limited to numbers, strings, and literals here):

```
data Format : Type where
     Num : Format -> Format
     Str : Format -> Format
     Lit : String -> Format -> Format
     End : Format
```

This describes the expected input types. We can calculate a corresponding function type:

```
PrintfType : Format -> Type
PrintfType (Num f) = (i : Int) -> PrintfType f
PrintfType (Str f) = (str : String) -> PrintfType f
PrintfType (Lit str f) = PrintfType f
PrintfType End = String
```

A function which computes a type can be used anywhere that Idris expects a value of type `Type`. So, for the type of `printf`, we name the first argument `fmt`, and use it to compute the rest of the type of `printf`:

```
printf : (fmt : Format) -> PrintfType fmt
```

We can check the type of an expression, even using an as yet undefined function, at the Idris REPL. For example, a format corresponding to `"%d %s"`:

```
Main> :t printf (Num (Lit " " (Str End)))
printf (Num (Lit " " (Str End))) : Int -> String -> String
```

We will implement this via a helper function which accumulates a string:

```
printfFmt : (fmt : Format) -> (acc : String) -> PrintfType fmt
```

Idris has support for *interactive* development, via editor plugins and REPL commands, and we use *holes* extensively. An expression of the form `?hole` stands for an as yet unimplemented part of a program. This defines a top level function `hole`, with a type but no definition, which we can inspect at the REPL. So, we can write a partial definition of `printfFmt`:

```
printfFmt : (fmt : Format) -> (acc : String) -> PrintfType fmt
printfFmt (Num f) acc = ?printfFmt_rhs_1
printfFmt (Str f) acc = ?printfFmt_rhs_2
printfFmt (Lit str f) acc = ?printfFmt_rhs_3
printfFmt End acc = ?printfFmt_rhs_4
```

Then, if we inspect the type of `printfFmt_rhs_1` at the REPL, we can see the types of the variables in scope, and the expected type of the right hand side:

```
Example> :t printfFmt_rhs_1
   f : Format
   acc : String
------------------------------
printfFmt_rhs_1 : Int -> PrintfType f
```

So, a format specifier of `Num` means we need to write a function that expects an `Int`. For reference, the complete definition is given in Listing 1. As a final step (omitted here) we can write a C-style `printf` by converting a `String` to a `Format` specifier. Here we use compile time data – the format specifier – to calculate the rest of the type. In Section 4, we will see a similar idea used to calculate a type from data which is not known until *run time*.

Listing 1 The complete definition of `printf`

```
printfFmt : (fmt : Format) -> (acc : String) -> PrintfType fmt
printfFmt (Num f) acc = \i => printfFmt f (acc ++ show i)
printfFmt (Str f) acc = \str => printfFmt f (acc ++ str)
printfFmt (Lit str f) acc = printfFmt f (acc ++ str)
printfFmt End acc = acc

printf : (fmt : Format) -> PrintfType fmt
printf fmt = printfFmt fmt ""
```

2.3.2 Dependent Data Types

We define data types (such as `List a` and `Format` earlier) by giving explicit types for the *type constructor* and the *data constructors*. We are not limited to parameterising by types; type constructors can be parameterised by any value. The canonical example is a vector, `Vect`, a list with its length encoded in the type:

```
data Vect : Nat -> Type -> Type where
     Nil  : Vect Z a
     (::) : a -> Vect k a -> Vect (S k) a
```

`Nat` is the type of natural numbers, where `Z` stands for "zero" and `S` stands for "successor". It is represented by a machine integer at run time. As we noted in Section 2.1, lower case names in type definitions are type-level variables. So, when we define `append` on vectors...

```
append : Vect n a -> Vect m a -> Vect (n + m) a
```

. . . `n`, `a` and `m` are type-level variables. Note that we do not say "type variable" since they are not necessarily of type `Type`! Type-level variables are bound as *implicit arguments*. Written out in full, the type of `append` is:

```
append : {n : Nat} -> {m : Nat} -> {a : Type} ->
         Vect n a -> Vect m a -> Vect (n + m) a
```

Note: We will refine this when introducing multiplicities in Section 3.

The implicit arguments are concrete arguments to `append`, like the `Vect` arguments. Their values are solved by unification [30, 20]. Typically implicit arguments are only necessary at compile time, and unused at run time. However, we can use an implicit argument in a definition, since the names of the arguments are in scope in the right hand side:

```
length : {n : Nat} -> Vect n a -> Nat
length xs = n
```

One challenge with first-class types is in distinguishing those parts of programs which are required at run time, and those which can be erased. Tradtionally, this phase distinction has been clear: types are erased at run time, values preserved. But this correspondence is merely a coincidence, arising from the special (non first-class) status of types! As we see with `length` and `append`, sometimes an argument might be required (in `length`) and sometimes it might be erasable (in `append`). Idris 1 uses a constraint solving algorithm [41], which has been effective in practice, but has a weakness that it is not possible to tell from a definition's type alone which arguments are required at run time. In Section 3.2 we will see how quantitative type theory allows us to make a precise distinction between the run time relevant parts of a program and the compile time only parts.

2.4 Idris 2

Idris 2 is a new version of Idris, implemented in itself, and based on Quantitative Type Theory (QTT) as defined in recent work by Atkey [4] following initial ideas by McBride [29]. In QTT, each variable binding is associated with a *quantity* (or *multiplicity*) which denotes the number of times a variable can be used in its scope: either zero, exactly once, or unrestricted. We will describe these further shortly. Several factors have motivated the new implementation:

- In implementing Idris in itself, we have necessarily done the engineering required on Idris to implement a system of this scale. Furthermore, although it is outside the scope of the present paper, we can explore the benefits of dependently typed programming in implementing a full-scale programming language.
- A limitation of Idris 1 is that it is not always clear which arguments to functions and constructors are required *at run time*, and which are erased, even despite previous work [7, 41, 42]. QTT allows us to state clearly, in a type, which arguments are erased. Erased arguments are still relevant *at compile time*.
- There has, up to now, been no full-scale implementation of a language based on QTT which allows exploration of the possibilities of linear and dependent types.
- Purely pragmatically, Idris 1 has outgrown the requirements of its initial experimental implementation, and since significant re-engineering has been required, it was felt that it was time to re-implement it in Idris itself.

In the following sections, we will discuss the new features in Idris 2 which arise as a result of using QTT as the core: firstly, how to express *erasure* in the type system; and secondly, how to encode resource usage protocols using linearity.

3 Quantities in Types

The biggest distinction between Idris 2 and its predecessor is that it is built on Quantitative Type Theory (QTT) [29, 4]. In QTT, each variable binding (including function arguments) has a *multiplicity*. QTT itself takes a general approach to multiplicities, allowing any semiring. For Idris 2, we make a concrete choice of semiring, where a multiplicity can be one of:

- 0: the variable is not used at run time
- 1: the variable is used exactly once at run time
- ω: no restrictions on the variable's usage at run time

In this section, we describe the syntax and informal semantics of multiplicities, and discuss the most important practical applications: erasure, and linearity. The formal semantics are presented elsewhere [4]; here, we aim to describe the intuition. In summary:

- Variable bindings have multiplicities which describe how often the variable must be used within the scope of its binding.
- A variable is "used" when it appears in the body of a definition (that is, not a type declaration), in an argument position with multiplicity 1 or ω.
- A function's type gives the multiplicities the arguments have in the function's body.

Variables with multiplicity ω are truly unrestricted, meaning that they can be passed in argument positions with multiplicity 0, 1 or ω. A function which takes an argument with multiplicity 1 promises that it will not share the argument in the future; there is no requirement that it has not been shared in the past.

3.1 Syntax

When declaring a funcion type we can, optionally, give an explicit multiplicity of `0` or `1`. If an argument has no multiplicity given, it defaults to ω. For example, we can declare the type of an identity function which takes its polymorpic type *explicitly*, and mark it erased:

```
id_explicit : (0 a : Type) -> a -> a
```

If we give a partial definition of `id_explicit`, then inspect the type of the hole in the right hand side, we can see the multiplicities of the variables in scope:

```
id_explicit a x = ?id_explicit_rhs

Main> :t id_explicit_rhs
0 a : Type
  x : a
------------------------------
id_explicit_rhs : a
```

This means that `a` is not available at run time, and `x` is unrestricted at run time. If there is no explicit multiplicity shown, it is ω. A variable which is not available at run time can only be used in argument positions with multiplicity `0`. Implicitly bound arguments are also given multiplicity `0`. So, in the following declaration of `append`...

```
append : Vect n a -> Vect m a -> Vect (n + m) a
```

...`n`, `a` and `m` have multiplicity `0`. In Idris 2, therefore, unlike Idris 1, the declaration is equivalent to writing:

```
append : {0 n : Nat} -> {0 m : Nat} -> {0 a : Type} ->
         Vect n a -> Vect m a -> Vect (n + m) a
```

The default multiplicities for arguments are, therefore:

- If you explicitly write the binding, the default is ω.
- If you omit the binding (e.g. in a type-level variable), the default is 0.

This design choice, that type-level variables are by default erased, is primarily influenced by the common usage in Idris 1, that implicit type-level variables are typically compile-time only. As a result, the most common use cases involve the most lightweight syntax.

3.2 Erasure

The multiplicity `0` gives us control over whether a function argument – `Type` or otherwise – is used at run time. This is important in a language with first-class types since we often parameterise types by values in order to make relationships between data explicit, or to make assumptions explicit. In this section, we will consider two examples of this, and see how multiplicity 0 allows us to control which data is available at run time.

3.2.1 Example 1: Vector length

We have seen how a vector's type includes its length; we can use this length at run time, even though it is part of the type, provided that it has non-zero multiplicity:

```
length : {n : Nat} -> Vect n a -> Nat
length xs = n
```

With this definition, the length `n` is available at run time, since `{n : Nat}` is an explicitly written binding so has default multiplicity ω. This has a run time cost, in that `n` is passed to the function at run time, as well as potentially the cost of computing the length elsewhere. If we want the length to be erased, we would need to recompute it in the `length` function:

```
length : Vect n a -> Nat
length [] = Z
length (_ :: xs) = S (length xs)
```

The type of `length` in each case makes it explicit whether or not the length of the `Vect` is available. Let us now consider a more realistic example, using the type system to ensure soundness of a compressed encoding of a list.

3.2.2 Example 2: Run-length Encoding of Lists

Run-length encoding is a compression scheme which collapses sequences (runs) of the same element in a list. It was originally developed for reducing the bandwidth required for television pictures [39], and later used in image compression and fax formats, among other things.

We will define a data type for storing run-length encoded lists, and use the type system to ensure that the encoding is sound with respect to the original list. To begin, we define a function which constructs a list by repeating an element a given number of times. We will need this for explaining the relationship between compressed and uncompressed data.

```
rep : Nat -> a -> List a
rep Z x = []
rep (S k) x = x :: rep k x
```

Using this, and the concatenation operator for `List` (`++`, which is defined like `append`), we can describe what it means to be a run-length encoded list:

```
data RunLength : List ty -> Type where
     Empty : RunLength []
     Run : (n : Nat) -> (x : ty) -> (rle : RunLength more) ->
           RunLength (rep (S n) x ++ more)
```

- `Empty` is the run-length encoding of the empty list `[]`
- Given a length `n`, an element `x`, and an encoding `rle` of the list `more`, `Run n x rle` is the encoding of the list `rep (S n) x ++ more`. That is, $n + 1$ copies of `x` followed by `more`.

We use `S n` to ensure that `Run` always increases the length of the list, but otherwise we make no attempt (in the type) to ensure that the encoding is optimal; we merely ensure that the encoding is sound with respect to the encoded list. Let us try to write a function which uncompresses a run-length encoded list, beginning with a partial definition:

```
uncompress : RunLength {ty} xs -> List ty
uncompress rle = ?uncompress_rhs
```

Note: The `{ty}` syntax gives an explicit value for the implicit argument to `RunLength`. This means that the `ty` argument to RunLength, and hence the element type of `xs`, is the same type as the element type of the list returned by `uncompress`.

Like our initial implementation of `length` on `Vect`, we might be tempted to return `xs` directly, since the index of the encoded list gives us the original uncompressed list. However, if we check the type of `uncompress_rhs`, we see that `xs` has multiplicity 0 because it is not an explicit argument, so isn't available at run time:

```
 0 xs : List ty
   rle : RunLength xs
------------------------------
uncompress_rhs : List ty
```

This is a good thing: if the uncompressed list were available at run-time, there would have been no point in compressing it! We can still take advantage of having the uncompressed list available as part of the type, though, by refining the type of `uncompress`:

```
data Singleton : a -> Type where
     Val : (x : a) -> Singleton x

uncompress : RunLength xs -> Singleton xs
```

A value of type `Singleton x` has exactly one value, `Val x`. The type of `uncompress` expresses that the uncompressed list is guaranteed to have the same value as the original list, although it must still be reconstructed at run-time. We can implement it as follows:

```
uncompress : RunLength xs -> Singleton xs
uncompress Empty = Val []
uncompress (Run n x y) = let Val ys = uncompress y in
                             Val (x :: (rep n x ++ ys))
```

Aside: This implementation was generated by type-directed program synthesis [38], rather than written by hand, taking advantage of the explicit relationship given in the type between the input and the output. The definition of `RunLength` more or less directly gives the uncompression algorithm, so we should not need to write it again!

The `0` multiplicity, therefore, allows us to reason about values at compile time, without requiring them to be present at run time. Furthermore, QTT gives us a guarantee of erasure, as well as an explicit type-level choice as to whether an index is erased or not.

3.3 Linearity

An argument with multiplicity `0` is guaranteed to be erased at run time. Correspondingly, an argument with multiplicity `1` is guaranteed to be used exactly once. The intuition, similar to that of Linear Haskell [6], is that, given a function type of the form...

```
f : (1 x : a) -> b
```

...then, if an expression `f x` is evaluated exactly once, `x` is evaluated exactly once in the process. QTT is a new core language, and the combination of linearity with dependent types has not yet been extensively explored. Thus, we consider the multiplicity `1` to be experimental, and in general Idris 2 programmers can get by without it – nothing in the Prelude exposes an interface which requires it. Nevertheless, we have found that an important application of linearity is in controlling resource usage. In the rest of this section, we describe two examples of this. First, we show in general how linearity can prevent us duplicating an argument, which can be important if the argument represents an external resource; then, we give a more concrete example showing how the `IO` type described in Section 2.2 is implemented.

3.3.1 Example 1: Preventing Duplication

To illustrate multiplicity `1` in practice, we can try (and fail!) to write a function which duplicates a value declared as "use once", interactively, where `LPair` is a linear pair type:

```
dup : (1 x : a) -> LPair a a
dup x = ?dup_rhs
```

Inspecting the `dup_rhs` hole shows that we have:

```
 0 a : Type
 1 x : a
--------------------------------------
dup_rhs : LPair a a
```

So, `a` is not available at run-time, and `x` must be used exactly once in the definition of `dup_rhs`. We can write a partial definition, where `#` is the constructor of `LPair`:

```
dup x = x # ?second_x
```

However, if we check the hole `second_x` we see that `x` is not available, because there was only 1 and it has already been consumed:

```
 0 a : Type
 0 x : a
--------------------------------------
second_x : a
```

We see the same result if we try `dup x = (?second_x, x)`. If we persist, and try...

```
dup x = x # x
```

...then Idris reports "`There are 2 uses of linear name x`".

▶ Remark. As we noted earlier, only usages in the body of a definition count. This means we can still parameterise data by linear variables. For example, if we have an `Ordered` predicate on lists, we can write an `insert` function on ordered linear lists:

```
insert : a -> (1 xs: List a) -> (0 _ : Ordered xs) -> List a
```

The use in `Ordered` does not count, and since `Ordered` has multiplicity 0 it is erased at run time, so any occurrence of `xs` when building the `Ordered` proof also does not count.

3.3.2 Example 2: I/O in Idris 2

Like Idris 1 and Haskell, Idris 2 uses a parameterised type `IO` to describe interactive actions. Unlike the previous implementation, this is implemented via a function which takes an abstract representation of the outside world, of primitive type `%World`:

```
PrimIO : Type -> Type
PrimIO a = (1 x : %World) -> IORes a
```

The `%World` is consumed exactly once, so it is not possible to use previous worlds (after all, you can't unplay a sound, or unsend a network message). It returns an `IORes`:

```
data IORes : Type -> Type where
     MkIORes : (result : a) -> (1 w : %World) -> IORes a
```

This is a pair of a result (with usage ω), and an updated world state. The intuition for multiplicities in data constructors is the same as in functions: here, if `MkIORes result w` is evaluated exactly once, then the world `w` is evaluated exactly once. We can now define `IO`:

```
data IO : Type -> Type where
     MkIO : (1 fn : PrimIO a) -> IO a
```

There is a primitive `io_bind` operator (which we can use to implement >>=), which guarantees that an action and its continuation are executed exactly once:

```
io_bind : (1 act : IO a) -> (1 k : a -> IO b) -> IO b
io_bind (MkIO fn) = \k => MkIO (\w => let MkIORes x' w' = fn w
                                      MkIO res = k x' in res w')
```

The multiplicities of the `let` bindings are inferred from the values being bound. Since `fn w` uses `w`, which is required to be linear from the type of `MkIO`, `MkIORes x' w'` must itself be linear, meaning that `w'` must also be linear. This implementation of `IO` is similar to the Haskell approach [37], with two differences:

1. The `%World` token is guaranteed to be consumed exactly once, so there is a type-level guarantee that the outside world is never duplicated or thrown away.
2. There is no built-in mechanism for exception handling, because the type of `io_bind` requires that the continuation is executed exactly once. So, in `IO` primitives, we must be explicit about where errors can occur. One can, however, implement higher level abstractions which allow exceptions if required.

Linearity is, therefore, fundamental to the implementation of `IO` in Idris 2. Fortunately, none of the implementation details need to be exposed to application programmers who are using `IO`. However, once a programmer has an understanding of the combination of linear and dependent types, they can use it to encode and verify more sophisticated APIs.

4 Linear Resource Usage Protocols

The `IO` type uses a linear resource representing the current state of the outside world. But, often, we need to work with other resources, such as files, network sockets, or communication channels. In this section, we introduce an extension of `IO` which allows creating and updating linear resources, and show how to use it to implement a resource usage protocol for an automated teller machine (ATM).

4.1 Initialising Linear Values

In QTT, multiplicities are associated with *binders*, not with return values or types. This is a design decision of QTT, rather than of Idris, and has the advantage that we can use a type linearly or not, depending on context. We can write functions that create values to be used linearly by using *continuations*, for example, to create a new linear array:

```
newArray : (size : Int) -> (1 k : (1 arr : Array t) -> a) -> a
```

The array must be used exactly once in the scope of `k`, and if this is the only way of constructing an `Array`, then all arrays are guaranteed to be used linearly, so we can have in-place update. As a matter of taste, however, we may not want to write programs with explicit continuations. Fortunately, `do` notation can help; recall the bind operator for `IO`:

```
(>>=) : IO a -> (a -> IO b) -> IO b
```

This allows us to chain an IO action and its continuation, and `do` notation gives syntactic sugar for translating into applications of >>=. Therefore, we can define an alternative >>= operator for chaining actions which return linear values. We will achieve this by defining a new type for capturing interactive actions, extending `IO`, and defining its bind operator.

4.2 Linear Interactive Programs

First, we define how many times the result of an operation can be used. These correspond to the multiplicities 0, 1 and ω:

```
data Usage = None | Linear | Unrestricted
```

We declare a data type `L`, which describes interactive programs that produce a result with a specific multiplicity. We choose a short name `L` since we expect this to be used often:

```
data L : {default Unrestricted use : Usage} ->
         Type -> Type where
```

In Section 2.3.2 we described implicit arguments, which are solved by unification. Here, `use` is a *default implicit* argument, and the `default Unrestricted` annotation means that if its value is not given explicitly, it will take a default value of `Unrestricted`.

Like `IO`, `L` provides the operators `pure` and >>=. However, unlike `IO`, they need to account for variable usage. One limitation of QTT is that it does not yet support quantity polymorphism, so we must provide separate `pure` operators for each quantity:

```
pure  : (  x : a) -> L         a
pure0 : (0 x : a) -> L {use=0} a
pure1 : (1 x : a) -> L {use=1} a
```

Idris translates integer literals using the `fromInteger` function. We have defined a `fromInteger` function that maps `0` to `None` and `1` to `Linear` which allows us to use integer literals as the values for the `use` argument.

The type of >>= is more challenging. In order to take advantage of `do`-notation, we need a single >>= operator for chaining an action and a continuation, but there are several possible combinators of variable usage. Consider:

- The action might return an erased, linear or unrestricted value.
- Correspondingly, the continuation must bind its argument at multiplicity 0, 1 or ω.

In other words, the type of the continuation to >>= depends on the usage of the result of the action. We can therefore take advantage of first-class types and *calculate* the continuation type. Given the usage of the action (`u_a`), the usage of the continuation (`u_k`) and the return types of each, `a` and `b`, we calculate:

```
ContType : (u_a : Usage) -> (u_k : Usage) -> Type -> Type -> Type
ContType None u_k a b = (0 _ : a) -> L {use=u_k} b
ContType Linear u_k a b = (1 _ : a) -> L {use=u_k} b
ContType Unrestricted u_k a b = a -> L {use=u_k} b
```

Then, we can write a type for >>= as follows:

```
(>>=) : {u_a : _} ->
        (1 _ : L {use=u_a} a) ->
        (1 _ : ContType u_act u_k a b) -> L {use=u_k} b
```

The continuation type is calculated from the usage of the first action, and is correspondingly needed in the implementation, so `u_a` is run time relevant. However, in practice it is removed by inlining. Fortunately, the user of `L` need not worry about these details. They can freely use `do`-notation and let the type checker take care of variable usage for them.

Finally, for developing linear wrappers for `IO` libraries, we allow lifting `IO` actions:

```
action : IO a -> L a
```

We use `action` for constructing primitives. Note that we will not be able to bypass any linearity checks this way, since it does not promise to use the `IO` action linearly, so we cannot pass any linear resources to an `action`. The implementation of `L` is via a well-typed interpreter [5], a standard pattern in dependently typed programming.

Note: `L` is defined in a library `Control.Linear.LIO`, distributed with Idris 2. In the library, its type is more general; `L : (io : Type -> Type) -> {use : Usage} -> Type -> Type`. This allows us to extend *any* monad with linearity, not just `IO`, but this generality is not necessary for the examples in this paper.

4.3 Example: An ATM State Machine

We can use linear types to encode the state of a resource, and implement operations in `L` to ensure that they are only executed when the resource is in the appropriate state. For example, an ATM should only dispense cash when a user has inserted their card and entered a correct PIN. This is a typical sequence of operations on an ATM:

- A user inserts their bank card.
- The machine prompts the user for their PIN, to check the user is entitled to use the card.
- If PIN entry is successful, the machine prompts the user for an amount of money, and then dispenses cash.

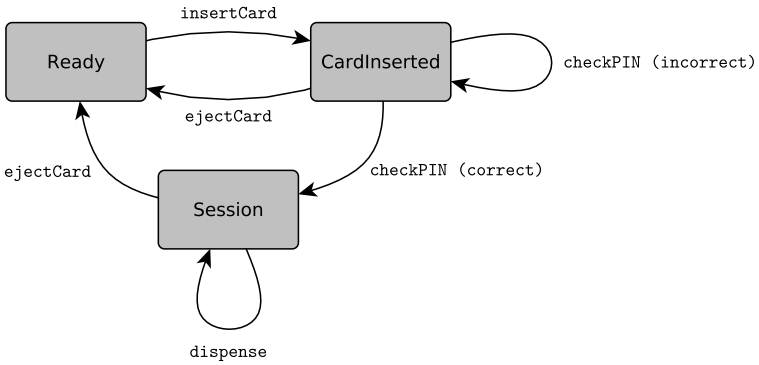


Figure 1 A state machine describing the states and operations on an ATM.

Figure 1 defines, at a high level, the states and operations on an ATM, showing when each operation is valid. We will define these operations as functions in the `L` type, using a linear reference to a representation of an ATM. We define the valid states of the ATM as a data type, then an ATM type which is parameterised by its current state, which is one of:

- `Ready`: the ATM is ready and waiting for a card to be inserted.
- `CardInserted`: there is a card inside the ATM but the PIN entry is not yet verified.
- `Session`: there is a card inside the ATM and the PIN has been verified.

```
data ATMState = Ready | CardInserted | Session
data ATM : ATMState -> Type
```

We leave the definition of `ATM` abstract. In practice, this is where we would need to handle implementation details such as how to access and update a user's bank account. For the purposes of this example, we are only interested in encoding the high level state transitions in types. We will need functions to initialise and shut down the reference:

```
initATM  : L {use=1} (ATM Ready)
shutDown : (1 _ : ATM Ready) -> L ()
```

`initATM` creates a linear reference to an ATM in the initial state, `Ready`, which must be used exactly once. Correspondingly, `shutDown` deletes the linear reference. Listing 2 presents the types of the remaining operations, implementing the transitions from Figure 1.

We have: user-directed state transitions, where the *programmer* is in control over whether an operation succeeds; general purpose operations, which do not change the state, and are part of the machine's user interface; and, machine-directed state transitions, where the *machine* is in control over whether an operation succeeds, for example the machine decides if PIN entry was correct.

User-directed state transitions

The `insertCard` card function takes a machine in the `Ready` state, and returns a new machine in the `CardInserted` state. The type ensures that we can only run the function with the machine in the appropriate state. For `dispense`, we need to satisfy the security property that the machine can only dispense money in a validated session. Thus, it has an input state of `Session`, and the session remains valid afterwards.

Listing 2 Operations on an ATM

```
data HasCard : ATMState -> Type where
     HasCardPINNotChecked : HasCard CardInserted
     HasCardPINChecked : HasCard Session

data PINCheck = CorrectPIN | IncorrectPIN

insertCard : (1 _ : ATM Ready) -> L {use=1} (ATM CardInserted)
checkPIN : (1 _ : ATM CardInserted) -> (pin : Int) ->
           L {use=1}
             (Res PINCheck
                  (\res => ATM (case res of
                                     CorrectPIN => Session
                                     IncorrectPIN => CardInserted)))
dispense : (1 _ : ATM Session) -> L {use=1} (ATM Session)
getInput : HasCard st => (1 _ : ATM st) ->
                         L {use=1} (Res String (const (ATM st)))
ejectCard : HasCard st => (1 _ : ATM st) -> L {use=1} (ATM Ready)
message : (1 _ : ATM st) -> String -> L {use=1} (ATM st)
```

For `ejectCard`, it is only valid to eject the card if there is already a card in the machine. This is true in *two* states: `CardInserted` and `Session`. Therefore, we define a *predicate* on states which holds for states where there is a card in the machine:

```
data HasCard : ATMState -> Type where
     HasCardPINNotChecked : HasCard CardInserted
     HasCardPINChecked : HasCard Session
```

The type of `ejectCard` then takes an input of type `HasCard st`, which is a proof that the predicate holds for the machine's input state `st`.

```
ejectCard : HasCard st => (1 _ : ATM st) -> L {use=1} (ATM Ready)
```

The notation `HasCard st => ...`, with the `=>` operator, means that this is an *auto implicit* argument. Like implicits, and `default` implicits, these can be omitted. The type checker will attempt to fill in a value by searching the possible constructors. In this case, if `st` is `CardInserted`, the value will be `HasCardPINNotCheck`, and if it is `Session`, the value will be `HasCardPINChecked`. Otherwise, Idris will not be able to find a value, and will report an error. Auto implicits are also used for interfaces, corresponding to type classes in Haskell, although we do not use interfaces elsewhere in this paper.

General purpose operations

The `message` function displays a message to the user. Its type means that we can display a message no matter the machine's state. Nevertheless, since an ATM is linear, we must return a new reference. The `getInput` function reads input from the user, using the machine's keypad, provided that there is a card in the machine. Again, this needs to return a new reference, along with the input. `Res` is a *dependent pair* type, where the first item is unrestricted, and the second item is linear with a type computed from the value of the first element. We describe this below in the context of `checkPIN`.

Machine-directed state transitions

The most interesting case is `checkPIN`. In the other functions, the programmer is in control of when state transitions happen, but in this case, the transition may or may not succeed, depending on whether the PIN is correct. To capture this possibility, we return the result in a dependent pair type, `Res`, defined as follows in the Idris Prelude:

```
data Res : (a : Type) -> (a -> Type) -> Type where
     (#) : (val : a) -> (1 r : t val) -> Res a t
```

This pairs a value, `val`, with a linear resource whose type is computed from the value. This can be illustrated with a partial ATM program:

```
runATM : L ()
runATM = do m <- initATM
            m <- insertCard m
            ok # m <- checkPIN m 1234
            ?whatnow
```

This program initialises an ATM, inserts a card, then checks whether the card has the PIN 1234. Checking the PIN returns a `Res`, which we deconstruct, then we can inspect the hole `?whatnow` to see where to go from here:

```
   ok : PINCheck
 1 m : ATM (case ok of { CorrectPIN => Session
                         IncorrectPIN => CardInserted })
------------------------------
whatnow : L ()
```

So, we have the result of the `PINCheck`, and an updated `ATM`, but we will only know the state of the `ATM`, and hence be able to make progress in the protocol, if we actually check the returned result! We cannot know statically what the next state of the machine is going to be, but using first class types, we *can* statically check that the necessary dynamic check is made. We could also use a sum type, such as `Either`, for the result of the PIN check, e.g.

```
checkPIN : (1 _ : ATM CardInserted) -> (pin : Int) ->
           L {use=1} (Either (ATM CardInserted) (ATM Session))
```

This would arguably be simpler. On the other hand, by returning an `ATM` with an as yet unknown state, we can still run operations on the `ATM` such as `message` even before resolving the state. This might be useful for diagnostics, or for user feedback, for example. Listing 3 shows one possible ATM protocol implementation, displaying a message before checking the PIN, then dispensing cash if the PIN was valid. Note that the protocol also requires the card to have been ejected before the machine is shut down.

Aside: Linearity and exceptions do not mix well, since when we catch an exception, we need to know what state the machine was in at the point it was thrown in order to clean up effectively. On the other hand, if we have to check every result as in Listing 3, we will end up with a lot of nested `case` blocks, which are hard to read. As a compromise, Idris provides a pattern matching bind notation [10], which allows us to code to a "happy path" and deal with alternatives as they arise. For example:

```
runATM : L ()
runATM = do m <- initATM
            m <- insertCard m
            CorrectPIN # m <- checkPIN m 1234
               | IncorrectPIN # m => ?failure
            ?success
```

Listing 3 Example ATM protocol implementation

```
runATM : L ()
runATM = do m <- initATM
            m <- insertCard m
            ok # m <- checkPIN m 1234
            m <- message m "Checking PIN"
            case ok of
                 CorrectPIN => do m <- dispense m
                                  m <- ejectCard m
                                  shutDown m
                 IncorrectPIN => do m <- ejectCard m
                                    shutDown m
```

The "happy path" is that the PIN was entered correctly. The alternative we need to handle is the `IncorrectPIN` case, which we can handle in a similar manner to Listing 3.

5 Session Types via QTT

To illustrate how we can use quantities on a more substantial example, let us consider how to use them to implement session types. Session types [21, 22] give types to communication channels, allowing us to express exactly *when* a message can be sent on a channel, ensuring that communication protocols are implemented completely and correctly. There has been extensive previous work on defining calculi for session types[3]. In Idris 2, the combination of linear and dependent types means that we can implement session types directly:

- **Linearity** means that a channel can only be accessed once, and once a message has been sent or received on a channel, the channel is in a new state.
- **Dependent Types** give us a way of describing protocols at the type level, where progress on a channel can change according to values sent on the channel.

A complete implementation of session types would be a paper in itself, so we limit ourselves to dyadic session types in concurrent communicating processes. We assume that functions are *total*, so processes will not terminate early and communication will always succeed. In a full library, dealing with *distributed* as well as *concurrent* processes, we would also need to consider failures such as timeouts and badly formed messages [18].

The key idea is to parameterise channels by the actions which will be executed on the channel – that is, the messages which will be sent and received – and to use channels linearly. We declare a `Channel` type as follows:

```
data Actions : Type where
     Send : (a : Type) -> (a -> Actions) -> Actions
     Recv : (a : Type) -> (a -> Actions) -> Actions
     Close : Actions

data Channel : Actions -> Type
```

[3] A collection of implementations is available at `http://groups.inf.ed.ac.uk/abcd/session-implementations.html`

Internally, `Channel` contains a message queue for bidirectional communication. Listing 4 shows the types of functions for initiating sessions, and sending and receiving messages. In the type of `send`, we see that to send a value of type `ty` we must have a channel in the state `Send ty next`, where `next` is a function that computes the rest of the protocol. The type of `recv` shows that we compute the rest of the protocol by inspecting the value received. We initiate concurrent sessions with `fork`, and will discuss the details of this shortly.

Listing 4 Initiating and executing concurrent sessions

```
send : (1 chan : Channel (Send ty next)) -> (val : ty) ->
       L {use=1} (Channel (next val))
recv : (1 chan : Channel (Recv ty next)) ->
       L {use=1} (Res ty (\val => Channel (next val)))
close : (1 chan : Channel Close) -> L ()
fork : ((1 chan : Server p) -> L ()) -> L {use=1} (Client p)
```

First, let us see how to describe dyadic protocols such that a *client* and *server* are guaranteed to be synchronised. We describe protocols via a *global* session type:

```
data Protocol : Type -> Type where
     Request : (a : Type) -> Protocol a
     Respond : (a : Type) -> Protocol a
     (>>=) : Protocol a -> (a -> Protocol b) -> Protocol b
     Done : Protocol ()
```

A protocol involves a sequence of `Request`s from a client to a server, and `Response`s from the server back to the client. For example, we could define a protocol (Listing 5) in which a client sends a `Command` to either `Add` a pair of `Int`s or `Reverse` a `String`.

Listing 5 A global session type describing a protocol where a client can request either adding two `Int`s or reversing a `String`

```
data Command = Add | Reverse

Utils : Protocol ()
Utils = do cmd <- Request Command
           case cmd of
                Add => do Request (Int, Int)
                          Respond Int
                          Done
                Reverse => do Request String
                              Respond String
                              Done
```

`Protocol` is a DSL for describing communication patterns. Embedding it in a dependently typed host language gives us dependent session types for free, as we will see in more detail at the end of this section. We use the embedding to our advantage in a small way, by having the protocol depend on `cmd`, the command sent by the client. We can write functions to calculate the protocol for the client and the server:

```
AsClient, AsServer : Protocol a -> Actions
```

We omit the definitions, but each translates `Request` and `Response` directly to the appropriate `Send` or `Receive` action. We can see how `Utils` translates into a type for the client side by running `AsClient Utils`:

```
Send Command (\res => (ClientK
    (case res of
          Add => Request (Int, Int) >>= \_ =>
                 Respond Int >>= \_ Done
          Reverse => Request String >>= \_ =>
                     Respond String >>= \_ Done)
```

Most importantly, this shows us that the first client side operation must be to send a `Command`. The rest of the type is calculated from the command which is sent; `ClientK` is internal to `AsClient` and calculates the continuation of the type. Using these, we can define the type for `fork`.

```
Client, Server : Protocol a -> Type
Client p = Channel (AsClient p)
Server p = Channel (AsServer p)

fork : ((1 chan : Server p) -> L ()) -> L {use=1} (Client p)
```

The type of `fork` ensures that the client and the server are working to the same protocol, by calculating the channel type of each from the same global protocol. Since each channel is linear, both ends of the protocol must be run to completion.

Listing 6 An implementation of a server for the `Utils` protocol

```
utilServer : (1 chan : Server Utils) -> L ()
utilServer chan
    = do cmd # chan <- recv chan
         case cmd of
              Add => do (x, y) # chan <- recv chan
                        chan <- send chan (x + y)
                        close chan
              Reverse => do str # chan <- recv chan
                            chan <- send chan (reverse str)
                            close chan
```

Listing 6 shows a complete implementation of a server for the `Utils` protocol. However, we do not typically write a complete implementation in one go. Idris 2's support for *holes* means that it is more convenient to write the server incrementally, in a type-driven way. We begin with just a skeleton definition, and look at the hole for the right hand side:

```
utilServer : (1 chan : Server Utils) -> L ()
utilServer chan = ?utilServer_rhs

 1 chan : Channel (Recv Command (\res => ... ))
--------------------------------------
utilServer_rhs : L ()
```

Therefore, the first action on `chan` must be to receive a `Command`:

```
utilServer : (1 chan : Server Utils) -> L ()
utilServer chan
    = do cmd # chan <- recv chan
         ?utilServer_rhs

   cmd : Command
 1 chan : Channel (ServerK (case cmd of ...) ...)
------------------------------
utilServer_rhs : L ()
```

We elide the full details of the type of `chan` at this stage, but at the top level it suggests that we can make progress by a `case` split on `cmd`:

```
utilServer : (1 chan : Server Utils) -> L ()
utilServer chan
    = do cmd # chan <- recv chan
         case cmd of
              Add => ?process_add
              Reverse => ?process_reverse
```

We make essential use of dependent `case` here, in that both branches have a different type which is computed from the value of the scrutinee `cmd`, similarly to `PrintfType` in Section 2.3.1. Now, for each of the holes `process_add` and `process_reverse` we see more concretely how the protocol should proceed. e.g. for `process_add`:

```
   1 chan : Channel (Recv (Int, Int) (\res =>
                    (Send Int (\res => Close))))
     cmd : Command
  -------------------------------------
  process_add : L ()
```

This shows we have to receive a pair of `Int`s, then send an `Int`. Programming is thus a *dialogue* with the type checker. Rather than trying to work out the complete program, with increasing frustration as the type checker rejects our attempts, we write the program step by step, and ask the type checker for more information on the variables in scope and the required result.

Dependent Session Types

We have the full language available at the type level, and we have already used this to our advantage in the definition of `Utils`, by using a `case` expression to choose how the protocol proceeds based on a sent value. This is a *dependent* session type, in that the protocol depends on run time information, although we have only used this to encode a form of choice. More interestingly, we can write functions which dynamically construct protocol descriptions. For example, the following function describes a server which sends back `n` values of type `a`:

```
GetN : (n : Nat) -> (a : Type) -> Protocol ()
GetN Z a = Done
GetN (S l) a = do Respond a
                  GetN l a
```

We can use this, for example, in a protocol in which a client sends a `Nat`, and the server responds with that many `String`s:

```
DepProto : Protocol ()
DepProto = do num <- Request Nat
              GetN num String
```

Listing 7 shows one possible implementation of a client for this session type, which sends a value, then receives exactly that many `String`s on the channel.

Listing 7 An implementation of a client using a dependent session type

```
depClient : (1 chan : Client DepProto) -> Nat -> L IO ()
depClient chan k
    = do chan2 <- send chan k
         getN k chan2
  where
    getN : (x : Nat) -> (1 chan : Client (GetN x String)) -> L IO ()
    getN 0 chan = close chan
    getN (S k) chan = do str # chan <- recv chan
                         putStrLn str
                         getN k chan
```

By embedding our session types implementation in a language with linear and dependent types, we need *no* extensions for dependent session types: the same machinery works for both standard and dependent sessions, thanks to the features of the host language.

Extension: Sending Channels over Channels

A useful extension is to allow a server to start up more worker processes to handle client requests. This would require sending the server's `Channel` endpoint to the worker process. However, we cannot do this with `send` as it stands, because the value sent must be of multiplicity ω, and the `Channel` is linear. One way to support this would be to refine `Protocol` to allow flagging messages as linear, then add:

```
send1 : (1 chan : Channel (Send1 ty next)) -> (1 val : ty) ->
        L {use=1} (Channel (next val))
```

This takes advantage of QTT's ability to parameterise types by linear variables like `val` here. A worker protocol, using the `Utils` protocol above, could then be described as follows, where a server forks a new worker process and immediately sends it the communication `Channel` for the client:

```
MakeWorker : Protocol ()
MakeWorker = do Request1 (Server Utils); Done
```

We leave full details of this implementation for future work. It is, nevertheless, a minor adaptation of the session types library.

6 Related Work

Substructural Types

Linear types [45] and other substructural type systems have several applications, e.g. verifying unique access to external resources [17] and as a basis for session types [21]. These applications typically use domain specific type systems, rather than the generality which would be given by

full dependent types. There are also several implementations of linear or other substructural type systems in functional languages [44, 36, 16, 33]. While these languages do not have full dependent types, Granule [36] allows many of the same properties to be expressed with a sophisticated notion of graded types which allows quantitative reasoning about resource usage, and work is in progress to add dependent types to Granule [32, 13]. ATS [40] is a functional language with linear types with support for theorem proving, which allows reasoning about resource usage and low level programming. An important mainstream example of the benefit of substructural type systems is Rust[4] [24] which guarantees memory safety of imperative programs without garbage collection or any run time overhead, and is expressive enough to implement session types [23].

Historically, combining linear types and dependent types in a fully general way – with first-class types, and the full language available at the type level – has been a difficult problem, primarily because it is not clear whether to count variable usages in types. The problem can be avoided [26] by disallowing dependent linear functions or by limiting the form of dependency [19], but these approaches limit expressivity. For example, we may still want to reason about linear variables which have been consumed. Or, as we saw at the end of Section 5, we may want to use a linear value as part of the computation of another type. Quantitative Type Theory [4, 29], allows full dependent types with no restrictions on whether variables are used in types or terms, by checking terms at a specific multiplicity.

Erasure

While linearity has benefits in allowing reasoning about effects and resource usage, one of the main motivations for using QTT is to give a clear semantics for erasure in the type system. We distinguish *erasure* from *relevance*, meaning that erased arguments are still relevant during type-checking, but erased at run time. Early approaches in Idris include the notion of "forced arguments" and "collapsible data types" [7], which give a predictable, if not fully general, method for determining which arguments can be erased. Idris 1 uses a whole program analysis [42], partly inspired by earlier work on Erasure Pure Type Systems [31] to determine which arguments can be erased, which works well in practice but doesn't allow a programmer to require specific arguments to be erased, and means that separate compilation is difficult. The problem of what to erase also exists in Haskell to some extent, even without full dependent types, when implementing zero cost coercions [46]. Our experience of the 0 multiplicity of QTT so far is that it provides the cleanest solution to the erasure problem, although we no longer infer which other arguments can be erased.

Reasoning about Effects

One of the motivations for using QTT beyond expressing erasure in types is that it provides a core language which allows reasoning about external resource usage. Previous work on reasoning about effects and resources with dependent types has relied on indexed monads [3, 27] or embedded DSLs for describing effects [9]. These are effective, but generally difficult to compose; even if we can compose effects in a single EDSL, it is hard to compose multiple EDSLs, especially when parameterised with type information. Other successful approaches such as Hoare Type Theory [34] are sufficiently expressive, but difficult to apply in everyday programming. Having linear types in the core language means that tracking state changes, which we have previously had to encode in a state-tracking monad, is now possible directly in the language. We can compose multiple resources by using multiple linear arguments.

[4] `https://rust-lang.org/`

Combining dependent and linear types, along with protocol descriptions in `L`, gives us similar power to Typestate [1, 48], in that we can use dependency to capture the state of a value in its type, and linearity to ensure that it is always used in a valid state. First-class types gives us additional flexibility: we can reason about state changes which are only known at run-time, such as checking a PIN in an ATM.

Session Types

In Section 5 we gave an example of using QTT to implement Dyadic Session Types [21]. In previous work [11] Idris has been experimentally extended with uniqueness types, to support verification of concurrent protocols. However, this earlier system did not support erasure, and as implemented it was hard to combine unique and non-unique references. Our experience with QTT is that its approach to linearity, with multiplicities on the binders rather than on the types, is much easier to combine with other non-linear programs.

Given linearity and dependent types, we can already have dependent session types, where, for example, the progress of a session depends on a message sent earlier. Thus, the embedding gives us label-dependent session types [43] with no additional cost. Previous work in exploring value-dependent sessions in a dependently typed language [15] is directly expressible using linearity in Idris 2. We have not yet explored further extensions to session types, however, such as multiparty session types [22], dealing with exceptions during protocol execution [18] or dealing with errors in transmission in distributed systems.

7 Conclusions and Further Work

Implementing Idris 2 with Quantitative Type Theory in the core has immediately given us a lot more expressivity in types than Idris 1. For most day to day programming tasks, expressing erasure at the type level is the most valuable user-visible new feature enabled by QTT, in that it is unambiguous which function and data arguments will be erased at run time. Erasure has been a difficulty for dependently typed languages for decades and until recently has been handled in partial and unsatisfying ways (e.g. [12]). Quantitative Types, and related recent work [42], are the most satisfying so far, in that they give the programmer complete control over what is erased at run time. In future, we may consider combining QTT with inference for additional erasure [42].

The 1 multiplicity enables programming with full linear dependent types. Therefore reasoning about resources, which previously required heavyweight library implementations, is now possible directly, in pure functions. We have also seen, briefly, that quantities give more information when inspecting the types of holes. More expressive types, with interactive editing tools. make programming a *dialogue* with the machine, rather than an exercise in frustration when submitting complete (but wrong!) programs to the type checker.

We have often found full dependent types, where a type is a first class language construct, to be extremely valuable in developing libraries with expressive interfaces, even if the programs which use those libraries do not use dependent types much. The `L` type for embedding linear protocols is an example of this, in that it allows a programmer to express precisely not only *what* a function does, but also *when* it is allowed to do it. It is important that the type system remains accessible to programmers, however. Dependent and linear types are powerful concepts, and without care in library design, can be hard to use. However, they don't have to be: they are based on concepts that programmers routinely understand and use, such as using a variable once and making assumptions about the relationships between data. A challenge for language and tool designers is to find the right syntax and feedback mechanisms, so that powerful verification tools are within reach of all software developers.

While we have already found many benefits of being able to express quantities in types, we have only just begun exploring, and have encountered some limitations in the theory which we hope to address, perhaps adapting ideas from related work [13]. Most importantly, we would like to express *polymorphic* quantities. This may, for example, help give an appropriate type to >>= taking into account that some monads guarantee to execute the continuation exactly once, but others need more flexibility. Similarly, like Granule [36], we may find it useful to use quantities other than `0` and `1`, and the theory behind QTT supports this.

We have not discussed performance in this paper, but for an interactive system it is vital, and will be a primary concern in the near future. Following [25], Idris 2 minimises substitution of unification solutions. Initial results are promising: Idris 2 is now self-hosting, and builds itself in around 90 seconds[5]. We are using the interactive development tools, especially holes, in developing Idris itself.

Finally, an important application of reasoning about linear resource usage is in implementing communication and security protocols correctly. The `Protocol` type in Section 5 provides a preliminary example which demonstrates the possibilities, but realistically it will need to handle timeouts, exceptions and more sophisticated protocols. Implementing these protocols correctly is difficult and error prone, and errors lead to damaging security problems[6]. But in describing a session type, we have explained a protocol in detail, and the machine calculates a lot of information about how the protocol proceeds. We should not let the type checker keep this information to itself! Thus, interactive programming of protocols based on linear resource usage gives a foundation for secure programming.

References

1 J Aldrich, J Sunshine, D Saini, and Z Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1015–1012, 2009. URL: `http://dl.acm.org/citation.cfm?id=1640073`.

2 Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-Scope Safe Programs and their Proofs. In *CPP*, pages 195–207, 2017.

3 Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335, 2009. `doi:10.1017/S095679680900728X`.

4 Robert Atkey. The syntax and semantics of quantitative type theory. In *LICS 2018*, 2018. `doi:10.1145/3209108.3209189`.

5 L. Augustsson and M. Carlsson. An exercise in dependent types: A well-typed interpreter. In *In Workshop on Dependent Types in Programming, Gothenburg*. Citeseer, 1999. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.2895`.

6 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL):5:1–5:29, December 2017. `doi:10.1145/3158093`.

7 Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.

8 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, September 2013.

9 Edwin Brady. Resource-dependent algebraic effects. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming (TFP '14)*, volume 8843 of *LNCS*. Springer, 2014.

[5] Dell XPS 13 Laptop, running Ubuntu 18.03 LTS

[6] e.g. `https://www.imperialviolet.org/2014/02/22/applebug.html`

10 Edwin Brady. Resource-dependent algebraic effects. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*, volume 8843 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2014. doi:10.1007/978-3-319-14675-1_2.

11 Edwin Brady. Type-driven development of concurrent communicating systems. *Computer Science*, 18(3), 2017.

12 Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs (TYPES 2003)*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2003.

13 Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie C. Weirich. A graded dependent type system with a usage-aware semantics (extended version). In *arXiv:2011.04070 [cs]*, January 2021. arXiv: 2011.04070. URL: http://arxiv.org/abs/2011.04070.

14 Nils Anders Danielsson. Total parser combinators. In *International Conference on Functional Programming (ICFP 2010)*, 2010. doi:10.1145/1932681.1863585.

15 Jan de Muijnck-Hughes, Edwin Brady, and Wim Vanderbauwhede. Value-dependent session design in a dependently typed language. In Francisco Martins and Dominic Orchard, editors, *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, volume 291 of *EPTCS*, pages 47–59, 2019. doi:10.4204/EPTCS.291.5.

16 Edsko de Vries, Rinus Plasmeijer, and David M Abrahamson. Uniqueness Typing Simplified. In *Implementation and Application of Functional Languages*, pages 201—-218, 2008.

17 Robert Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In David Schmidt, editor, *Programming Languages and Systems*, pages 204–218, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

18 Simon Fowler, Sam Lindley, J Garrett Morris, and Sara Decova. Exceptional Asynchronous Session Types: Session Types without Tiers. In *Principles of Programming Languages (POPL 2019)*, 2019.

19 Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. Linear dependent types for differential privacy. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '13*, page 357, 2013. doi:10.1145/2429069.2429113.

20 Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD Thesis, University of Strathclyde, 2013. URL: https://personal.cis.strath.ac.uk/adam.gundry/thesis/thesis-2013-07-24.pdf.

21 Kohei Honda. Types for dyadic interaction. In *CONCUR 1993 (International Conference on Concurrency Theory)*. Springer, 1993.

22 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Principles of Programming Languages (POPL 2008)*, 2008.

23 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session Types for Rust. In *WGP 2015 (Workshop on Generic Programming)*. ACM, 2015.

24 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), 2017. doi:10.1145/3158154.

25 András Kovács. Fast elaboration for dependent type theories, 2019. Talk at EU Types WG Meeting.

26 Neelakantan R Krishnaswami, Pierre Pradic, and Nick Benton. Integrating Dependent and Linear Types. In *Principles of Programming Languages (POPL 2015)*, 2015.

27 Conor McBride. Kleisli arrows of outrageous fortune, 2011.

28 Conor McBride. How to Keep Your Neighbours in Order. In *International Conference on Functional Programming (ICFP 2014)*, 2014.

29 Conor McBride. I got plenty o' nuttin'. In *A List of Successes that Can Change the World*, 2016.

30 Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 1992. URL: http://www.sciencedirect.com/science/article/pii/074771719290011R.

31 Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in pure type systems. In Roberto M. Amadio, editor, *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2008. doi:10.1007/978-3-540-78499-9_25.

32 Benjamin Moon, Harley Eades III, and Dominic Orchard. Graded Modal Dependent Type Theory. In *ESOP 2021*, 2020. arXiv: 2010.13163. URL: http://arxiv.org/abs/2010.13163.

33 J Garrett Morris. The Best of Both Worlds: Linear Functional Programming Without Compromise. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming - ICFP 2016*, pages 448–461, 2016. doi:10.1145/2951913.2951925.

34 Aleksander Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *International Conference on Functional Programming (ICFP 2008)*, pages 229—-240, 2008. doi:10.1145/1411204.1411237.

35 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.7934&rep=rep1&type=pdf.

36 Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.*, 3(ICFP), 2019. doi:10.1145/3341714.

37 Simon Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction, Marktoberdorf Summer School*, pages 47—-96, 2001.

38 Nadia Polikarpova, Ivan Kuraj, and Armando Solar-lezama. Program Synthesis from Polymorphic Refinement Types. *PLDI*, 2016. ISBN: 9781450342612.

39 A. H. Robinson and C. Cherry. Results of a prototype television bandwidth compression scheme. *Proceedings of the IEEE*, 55(3):356–364, 1967. doi:10.1109/PROC.1967.5493.

40 Rui Shi and Hongwei Xi. A linear type system for multicore programming in ATS. *Science of Computer Programming*, 78(8):1176–1192, 2013. doi:10.1016/j.scico.2012.09.005.

41 Matúš Tejiščák. A dependently typed calculus with pattern matching and erasure inference. *Proc. ACM Program. Lang.*, 4(ICFP):91:1–91:29, 2020. doi:10.1145/3408973.

42 Matúš Tejiščák. *Erasure in Dependently Typed Programming*. PhD thesis, University of St Andrews, 2020.

43 Peter Thiemann and Vasco T. Vasconcelos. Label-dependent session types. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371135.

44 Jesse a. Tov and Riccardo Pucella. Practical affine types. In *Principles of Programming Languages*, pages 447—-458, 2011. doi:10.1145/1925844.1926436.

45 Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

46 Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard A. Eisenberg. A role for dependent types in Haskell. *Proc. ACM Program. Lang.*, 3(ICFP):101:1–101:29, 2019. doi:10.1145/3341705.

47 Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. A specification for dependent types in Haskell. *Proc. ACM Program. Lang.*, 1(ICFP):31:1–31:29, 2017. doi:10.1145/3110275.

48 Roger Wolff, Jonathan Aldrich, Ronald Garcia, Roger Wolff, and Jonathan Aldrich. Foundations of Typestate-Oriented Programming. *Transactions on Programming Languages and Systems*, 36(4):1–44, 2014.

Multiparty Session Types for Safe Runtime Adaptation in an Actor Language

Paul Harvey
Rakuten Mobile Innovation Studio, Tokyo, Japan

Simon Fowler
School of Computing Science, University of Glasgow, Scotland, UK

Ornela Dardha
School of Computing Science, University of Glasgow, Scotland, UK

Simon J. Gay
School of Computing Science, University of Glasgow, Scotland, UK

Abstract

Human fallibility, unpredictable operating environments, and the heterogeneity of hardware devices are driving the need for software to be able to *adapt* as seen in the Internet of Things or telecommunication networks. Unfortunately, mainstream programming languages do not readily allow a software component to sense and respond to its operating environment, by *discovering*, *replacing*, and *communicating* with components that are not part of the original system design, while maintaining static correctness guarantees. In particular, if a new component is discovered at runtime, there is no guarantee that its communication behaviour is compatible with existing components.

We address this problem by using *multiparty session types with explicit connection actions*, a type formalism used to model distributed communication protocols. By associating session types with software components, the discovery process can check protocol compatibility and, when required, correctly replace components without jeopardising safety.

We present the design and implementation of EnsembleS, the *first* actor-based language with adaptive features and a static session type system, and apply it to a case study based on an adaptive DNS server. We formalise the type system of EnsembleS and prove the safety of well-typed programs, making essential use of recent advances in *non-classical* multiparty session types.

2012 ACM Subject Classification Software and its engineering → Concurrent programming languages

Keywords and phrases Concurrency, session types, adaptation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.10

Related Version *Full Version*: `https://arxiv.org/abs/2105.06973`

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.8`

Funding Supported by EPSRC grants EP/T014628/1 (STARDUST), EP/K034413/1 (ABCD), EP/L01503X/1 (CDT in Pervasive Parallelism), ERC Consolidator Grant Skye (682315), and by the EU HORIZON 2020 MSCA RISE project 778233 (BehAPI).

Acknowledgements Thanks to Phil Trinder for helpful comments and discussions, and to the anonymous reviewers for exceptionally detailed reviews.

1 Introduction

The era of single monolithic stand-alone computers has long been replaced by a landscape of heterogeneous and distributed computers and software applications. Technologies such as the IoT [56], self-driving cars [55], or autonomous networks [7] bring the new challenge of needing to successfully operate in face of ever-changing environments, technologies, devices, and human errors, necessitating the need to *adapt*.

Here, we define *dynamic self-adaptation* – hereafter referred to as *adaptation* – as the ability of a software component to sense and respond to its operating environment, by *discovering*, *replacing*, and *communicating* with other software components at runtime that are not part of the original system design [6, 52]. There are many examples of adaptive systems, as well as the mechanisms of adaptation they leverage, such as discovery [37], modularisation [27], dynamic code loading and migration [12, 24]. Commercially, Steam's in-home streaming system[1] enables video games to dynamically transfer their input/output across a range of devices. Academically, RE$^{\text{X}}$ [50] enables software to self-assemble predefined components, using machine learning to reconfigure the software in response to environmental changes.

Despite strong interest in adaption and substantial work on the mechanisms of adaptation, current programming languages either lack the capabilities to ensure that adaptation can be achieved safely and correctly, or they check correctness dynamically, resulting in runtime overheads which may not be acceptable for resource-constrained devices.

Specifically, if an adaptive system discovers new software components at runtime, these components must interact with the system in a purposeful manner. In concurrent and distributed systems, such interaction goes beyond a simple function call / return expressed with standard types and type systems: interaction involves complex *communication protocols* that constrain the sequence and type of data exchanged. For example, knowing that two components communicate integers and strings does not describe if or when they will be sent or received. In spite of growing interest in the topic, for example, the recent formation of the United Nations group considering *creative adaptation*[2], mainstream programming languages do not support the *specification* and *verification* of communication protocols in concurrent and distributed systems. In turn, errors are discovered late in the development process and potentially after deployment.

Even where all components are known statically, communication safety cannot be guaranteed: as an example, the RE$^{\text{X}}$ system's programming language specifies sequential call / return interfaces for components, but not communication protocols for concurrent components. The adaptation in the Steam in-home streaming system is even more limited, being restricted to detection of input/output devices from a set of compatible possibilities. In both cases, the adaptive aspects of the software have been defined and designed ahead of time, as opposed to being composed *on-demand* at runtime, leaving no scope for extending the system via runtime discovery and replacement.

This situation brings us to a key research question:

RQ: Can a programming language support *static (compile-time) verification* of safe runtime dynamic self-adaptation, i.e., *discovery, replacement and communication*?

The problem of static verification of safe communication is addressed by *multiparty session types* [29, 30, 31]. Multiparty session types (MPSTs) are a type formalism used to specify the *type*, *direction* and *sequence* of communication actions between two or more participants. Session types guarantee that software conforms to predefined communication protocols, rather than risking errors manifesting themselves at runtime.

There is already some work in the literature on adaptation and session types, but it does not answer our research question. We discuss related work in §6, but in brief, the state-of-the-art has some combination of the following limitations: theory for a formal model such as the π-calculus [13, 11, 19, 18], rather than a real-world programming language; omission of some aspects of adaptation, such as runtime discovery [32]; or verification by runtime monitoring [47, 49, 21], as opposed to static checking.

[1] `http://store.steampowered.com/streaming/`
[2] `https://www.itu.int/en/ITU-T/focusgroups/an/Pages/default.aspx`

Global protocol

```
1  global protocol Bookstore
2      (role Sell, role Buy1, role Buy2) {
3    book(string) from Buy1 to Sell;
4    book(int) from Sell to Buy1;
5    quote(int) from Buy1 to Buy2;
6    choice at Buy2 {
7      agree(string) from Buy2 to Buy1, Sell;
8      transfer(int) from Buy1 to Sell;
9      transfer(int) from Buy2 to Sell;
10   } or {
11     quit(string) from Buy2 to Buy1, Sell;
12   } }
```

Local protocol for Sell

```
1  local protocol Bookstore_Sell
2      (self Sell,role Buy1,role Buy2) {
3    book(string) from Buy1;
4    book(int) to Buy1;
5    choice at Buy2{
6      agree(string) from Buy2;
7      transfer(int) from Buy1;
8      transfer(int) from Buy2;
9    } or {
10     quit(string) from Buy2;
11 } }
```

Figure 1 Global and local protocols for Bookstore.

To answer our research question, we implement EnsembleS, the first actor language leveraging MPSTs to provide compile-time verification of safe dynamic runtime adaptation: we can statically guarantee that a discovered actor will comply with a communication protocol, and guarantee that replacing an actor's behaviour (e.g., to fix a bug) will not jeopardise communication safety. Key to our approach is the combination of the actor paradigm [28], for its process addressability and explicit message passing, with *explicit connection actions* [32] in multiparty session types, which allow discovered actors to be invited into a session.

Contributions. The overarching contribution of this work is the design, implementation, and formalisation of a language which supports dynamic self-adaptation while guaranteeing communication safety. We achieve this through a novel integration of an actor-based language and multiparty session types with explicit connection actions. Specifically, we introduce:

1. **EnsembleS and its compiler** (§ 3): we present an actor language, EnsembleS, which supports safe adaptable applications using MPSTs. Our framework supports:
   - MPST specifications, both standard and using explicit connection actions (§ 3.3);
   - MPSTs to provide guarantees of protocol compliance in runtime discovery (§ 3.4);
   - automatic generation of application code from MPSTs (§ 3.2)
2. **An adaptive DNS case study** (§ 4): using MPSTs and runtime discovery to show safe dynamic self-adaptation can be achieved in a non-trivial software service
3. **A core calculus for EnsembleS** (§ 5): we formalise EnsembleS and prove type safety and progress.

The formalism makes several technical contributions: it is the first actor-based calculus with statically-checked MPSTs; and it is the first *calculus* to provide a language design and semantics for explicit connection actions, which had previously only been explored at the type level. Our design requires exception handling in the style of Mostrous & Vasconcelos [45] and Fowler *et al.* [22], and the metatheory makes essential (and novel) use of *non-classical* multiparty session types [54].

The implementation and examples are available in the paper's companion artifact.

2 Multiparty Session Types

Multiparty session types [31] are a type formalism used to describe communication protocols in concurrent and distributed systems. An MPST describes communication among multiple software components or participants, by specifying the *type* and the *direction* of data exchanged, which is given as a sequence of send and receive actions.

```
1  explicit global protocol OnlineStore
2      (role Customer, role Store, role Courier) {
3    login(string) connect Customer to Store;
4    do Browse(Customer, Store, Courier);
5  }
6
7  aux global protocol Deliver
8      (role Customer, role Store, role Courier) {
9    address(string) from Customer to Store;
10   deliver(string) connect Store to Courier;
11   ref(int) from Courier to Store;
12   disconnect Courier and Store;
13   ref(int) from Store to Customer;
14   disconnect Store and Customer;
15 }
```

```
1  aux global protocol Browse
2      (role Customer, role Store, role Courier) {
3    item(string) from Customer to Store;
4    price(int) from Store to Customer;
5    choice at Customer {
6      do Browse(Customer, Store, Courier);
7    } or {
8      do Deliver(Customer, Store, Courier);
9    } or {
10     quit() from Customer to Store;
11     disconnect Store and Customer;
12   }
13 }
```

Figure 2 Global protocol for `OnlineStore`.

We first introduce MPSTs (formalised in § 5) via *Scribble* [57], a specification language for communicating protocols based on the theory of multiparty session types. We start with a *global type*, which describes the interactions among all communicating participants. Using the Scribble tool, a global protocol can be *validated*, guaranteeing its correctness, and then *projected* for each participant. Projection returns a *local type*, which describes communication actions from the viewpoint of that participant.

`Bookstore` example. Fig. 1 shows the classic `Bookstore` (also known as *Two-Buyer*) example, written in Scribble. We have three communicating participants (*roles*): two buyers `Buy1` and `Buy2`, and one seller `Sell`, where the buyers wish to buy a book from the seller. `Buy1` sends the title of the book of type **string** to `Sell` (line 3). Next, `Sell` sends the price of the book of type **int** to `Buy1` (line 4). At this stage, `Buy1` invites `Buy2` to share the cost of the book, by sending them a `quote` of type **int** that `Buy2` should pay (line 5). It is `Buy2`'s *internal choice* (line 6) to either `agree` (line 7), or `quit` the protocol (line 11). After agreement, both `Buy1` and `Buy2` `transfer` their `quote` to `Sell` (lines 8 and 9, respectively).

Projecting the `Bookstore` global protocol into each of the communicating participants returns their local protocols. Fig. 1 shows the local protocol for `Sell`; we omit `Buy1` and `Buy2` as they are similar. Note that the local protocol only includes actions relevant to `Sell`.

Explicit connection actions. The `Bookstore` protocol assumes that all roles are connected at the start of the session. This is undesirable when a participant is only needed for *part* of a session, or the identity of a participant depends on data exchanged in the protocol.

Consider Figure 2, which details the protocol for an online shopping service, inspired by the travel agency protocol detailed by Hu & Yoshida [32]. The protocol is organised as three subprotocols: `OnlineStore`, the entry-point; `Browse`, where the customer repeatedly requests quotes for items; and `Deliver`, where the store requests delivery from a courier. In contrast to `Bookstore`, each connection must be established explicitly (note that connect replaces from when initiating a connection).

Note in particular that `Courier` is only involved in the `Deliver` subprotocol. The store can therefore *choose* which courier to use based on, for example, the weight of the item or the customer's location. Furthermore, it is not necessary to involve the courier if the customer does not choose to make a purchase.

```
1  type Isnd is interface(out integer output)
2  type Ircv is interface(in integer input)
3
4  stage home {
5    actor sender presents Isnd {
6      value = 1;
7      constructor() {}
8      behaviour {
9        send value on output;
10       value := value + 1;
11   } }
12   actor receiver presents Ircv {
13     constructor() {}
14     behaviour{
15       receive data from input;
16       printString("\nreceived:␣");
17       printInt(data);
18   } }
19   boot {
20     s = new sender();
21     r = new receiver();
22     establish topology(s, r);
23   }
24 }
```

Figure 3 A simple EnsembleS program.

3 EnsembleS: An Actor Language for Runtime Adaptation

In this section, we present EnsembleS, a new session-typed actor-based language based on Ensemble [25, 26]. EnsembleS actors are addressable, single-threaded entities with share-nothing semantics, and communicate via message passing. However, differently from the classic definition of the actor model [28, 1], the communication model in EnsembleS is channel-based. EnsembleS supports both *static* and *dynamic* topologies:

Static Topologies All participants are present at the start of the session and remain involved for the duration of the session. This is based on traditional MPSTs [31].

Dynamic Topologies Participants can connect and disconnect during a session. This builds on the more recent idea of explicit connection actions [32].

3.1 EnsembleS: basic language features

An EnsembleS actor has its own private state and a single thread of control expressed as a `behaviour` clause, which is repeated until explicitly told to stop. Every actor executes within a `stage`, which represents a memory space. Actors do not share state, but instead communicate via message passing along half-duplex, simply-typed channels.

Fig. 3 shows a simple EnsembleS program which defines, instantiates and connects two actors, one of which sends increasing values to the other. The program defines two interfaces Isnd and Ircv, declaring an output and input channel respectively. The boot clause (lines 19–23) is executed first and creates an instance of each actor (lines 20–21), using the appropriate constructor (lines 7 and 13, respectively). This creates and begins executing new threads for each actor, which follow the logic of the relevant behaviour clause. Next, the boot clause binds the actor's channels together (line 22, discussed in §3.3). Once bound, the sender actor sends the contents of value on its channel, increments it, and goes back to the beginning of its behaviour loop (lines 8–11). The receiver actor waits for a message, binds the message to data, displays it, and returns to the top of its behaviour loop (lines 14–18). EnsembleS inherits Ensemble's support for runtime software adaptation actions [26]:

Discover The ability to *locate* an arbitrary actor or stage reference at runtime, given an `interface` and `query`.

Install Given an actor type, the ability to `spawn` it at a specified stage.

Migrate The ability for an executing actor to *move* to another stage.

Replace The ability to *replace* an executing actor A by a new instantiation of actor B, the latter continuing at the same stage as A, if A and B have the same `interface`.

Interact : Given an actor reference (either spawned, discovered, or communicated), the ability to *connect* to its channels at runtime and then communicate.

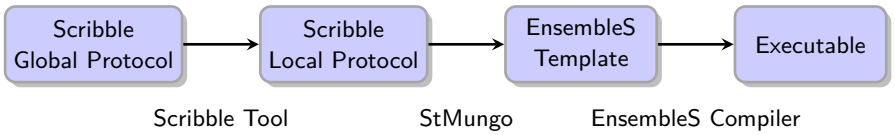


Figure 4 Automatic Actor Skeleton Generation Process.

We focus on the underlined actions and apply session types to guarantee communication safety. The reason for this choice is that discover, replace and interact are actions that modify *how* actors operate, whereas the other actions, install and migrate, affect *where* actors operate, but not their behaviour.

3.2 Session types in EnsembleS

A `session` type in EnsembleS represents a communication protocol for an actor, i.e., a local protocol (or local session type) validated and projected from a global session type.

We extend the StMungo [40] tool to generate EnsembleS template code that supports session types. Fig. 4 shows an overview of the actor template code generation from a global session type, and Fig. 5 shows an example of the generated code.

First, a developer defines a global session type in Scribble [57] (Fig. 4, first stage). The Scribble tool checks that the protocol is well-formed and valid according to MPST theory and *projects* the global protocol into local protocols for each participant (Fig. 4, second stage). For each local protocol, the StMungo tool produces (Fig. 4, third stage) *i*) the `session` type, *ii*) the **interface** and type definitions, and *iii*) the **actor** template. The generated code is parsed by the EnsembleS compiler, producing executable code (Fig. 4, fourth stage).

Let us now look at the `Buy1` local protocol, given in Fig. 1. Following the code generation process in Fig. 4, the EnsembleS template items *i*), *ii*) and *iii*) for `Buy1` correspond respectively to the code blocks starting in lines 3, 14, and 24 in Fig. 5.

The `Buy1` local protocol is translated as an EnsembleS `session` type in Fig. 5 (lines 3–12). It shows a sequence of send and receive actions (lines 4–6), followed by a choice at `Buy2` (lines 7–12), which determines the next set of communication actions.

Following session type specifications, EnsembleS channels define both the payload type and the `session` that this channel expects to interact with (lines 14–21, Fig. 5). The EnsembleS compiler uses this information to ensure that the `session` of each channel matches the `session` associated with the actor it is connected to.

An `actor` may `follow` a `session` type (line 24, Fig. 5). This tells the EnsembleS compiler that the logic within the `behaviour` clause of that actor must follow the communication protocol defined in the `session`.

It is important to note that the code generation in Fig. 4 is optional and the EnsembleS typechecker is independent of this process.

3.3 Channel connections: static and dynamic

If an actor follows a `session` type, then its channel connections must be *1-1*. This is the standard linearity requirement for session types: if there are multiple senders on one channel, then their messages can interfere and it is not possible to statically check that the session is followed correctly. EnsembleS avoids this problem by using a single channel for each message type between each pair of participants. For example, in Fig. 1, each of the three actors communicates strings and integers with both of the other actors. Because channels are unidirectional, each actor therefore has 8 channels: 2 to send strings and 2 to send integers to both other actors, and similarly 4 channels for receiving.

```
1  // FILE AUTOMATICALLY GENERATED
2  //************SESSIONS**************
3  type Buy1 is session(
4    book(string) to Sell;
5    book(int) from Sell;
6    quote(int) to Buy2;
7    choice at Buy2{
8      Choice0_agree(string) from Buy2;
9      transfer(int) to Sell;
10   }  or {
11     Choice0_quit(string) from Buy2;
12   } )
13 //***********INTERFACES*************
14 type Buy1I is interface(
15   out {Seller, string} toSell_string,
16   in {Seller, integer} fromSell_integer,
17   out {Buy2, integer} toBuy2_integer,
18   in {Buy2, Choice0} fromBuy2_agreequit,
19   in {Buy2, string} fromBuy2_string,
20   out {Sell, integer} toSell_integer,
21 )
22 //*************ACTORS***************
23 stage home{
24 actor Buy1A presents Buy1I follows Buy1 {
25  constructor() {}
26  behaviour {
27   payload1 = "";
28   send payload1 on toSell_string;
29   receive payload2 from fromSell_integer;
30   payload3 = 42;
31   send payload3 on toBuy2_integer;
32   // Receive choice from other actor
33   receive payload4 from fromBuy2_agreequit;
34   switch(payload4) {
35    case Choice0_agree:
36      receive payload5 from fromBuy2_string;
37      payload6 = 42;
38      send payload6 on toSell_integer;
39      break;
40    case Choice0_quit:
41      receive payload7 from fromBuy2_string;
42      break;
43   }
44 } }
45 // Omitted: Buy2A and SellA actors
46 boot {
47  buyer1 = new Buy1A();
48  buyer2 = new Buy2A();
49  seller = new SellA();
50  // other actors...
51  establish topology(buyer1,buyer2,seller);
52 } }
```

Figure 5 EnsembleS static session template.

Static connections. When using session types with *static* topologies, and all actors in the session are known from the beginning of the application, EnsembleS provides the `establish topology` statement to create the connections between the specified `session` actors (line 22, Fig. 3; line 51, Fig. 5). A compile-time error is generated if the topology is ill-defined (e.g., if the sessions do not compose or if the channels do not match).

Dynamic connections. EnsembleS supports reconfigurable channels and *dynamic* connections, via `link` and `unlink` statements. The `link` statement takes two references to actors which follow `session`s (line 5, top of Fig. 6), and connects all of the channels of the two specified actors such that the actors' `session`s match. A compile-time error is raised if the `session`s are incompatible. Conversely, the `unlink` statement disconnects (line 8).

3.4 Adaptation via discovery and replacement

EnsembleS supports runtime discovery of *local* or *remote* actor instances. As an example, in a sensor network, it may be desirable to connect to a sensor which has a battery level above a certain threshold. The EnsembleS query language allows us to define a query on non-functional properties (such as battery level or signal strength), as well as the channels exposed by an actor's interface. This ensures that any discovered actor has the correct number and type of channels, and satisfies user's preferences. To ensure that the discovered actor also obeys a declared protocol, EnsembleS uses `session` types in the discovery process. The green box in Fig. 6 shows how a `session` is used in the actor discovery process, and the yellow box shows how such actors are connected together. Runtime discovery does not appear in the `session` because it does not affect the communication behaviour of an actor.

EnsembleS also supports the replacement of executing actors, much like the hot-code swapping in Erlang [12]. The new actor must present the same interface as it *takes over* the channels of the actor being replaced at the location it was executing. Replacement happens at the beginning of an actor's `behaviour` loop. Replacement has many uses, such as updating,

Discovery and explicit connections

```
1 query alpha() { $serial==823 && $version<4; }
2 actor_s = discover(
3     Buyer1_interface, Buyer1_session, alpha());
4 if (actor_s[0].length > 1){
5   link me with actor_s[0];
6   msg = "book";
7   send msg on toB_string;
8   unlink Buyer1_session;
9 }
```

Replacement

```
1  // session and interface definitions
2  actor fastA presents accountingI
3         follows accountingSession{
4    constructor() {}
5    behaviour{
6      receive data on input;
7      quicksort(data);
8      send data on output;
9    }
10 }
11
12 actor slowA presents accountingI
13        follows accountingSession{
14   pS= new property[2] of property("",0);
15   constructor() {
16     pS[0]:= new property("serial",823);
17     pS[1]:= new property("version",2);
18     publish pS;
19   }
20   behaviour{
21     receive data on input;
22     bubblesort(data);
23     send data on output;
24 }  }
25
26 actor main presents mainI {
27  constructor() { }
28  behaviour {
29   // Find the slow actors matching query
30   actor_s = discover(accountingI,
31     accountingSession, alpha());
32   // Replace them with efficient versions
33   if(actor_s[0].length > 1) {
34     replace actor_s[0] with fastA();
35   }
36 } }
```

Figure 6 Session type-based adaptation.

changing, or extending some of the functionalities of existing software, and is particularly useful in embedded systems [33, 34]. The existing and new actors must follow the same `session` type, guaranteeing that replacement will not break existing actor interactions.

Fig. 6 (bottom) shows an example of a *main* actor searching for actors of type `slowA` (line 30), and replacing them with new actors of type `fastA` (line 34). The `slowA` actors are located by defining a query (line 1, top) over user-defined properties, which are published (lines 16–18). The discovery process is the same as above, but now the discovered actors are used for replacement rather than just communication.

3.5 Implementation

EnsembleS is implemented in C, and supports reference-counted garbage collection and exceptions. Applications are compiled to Java source code, and then to custom Java class files for use with a custom VM [10]. These applications can be executed on the desktop, parallel accelerators (e.g. GPUs), Raspberry Pi, Lego NXT, and Tmote Sky hardware platforms, and use a range of networking technologies.

Compact representations of session types are retained at runtime in order to support discovery. EnsembleS skeleton generation code is based on the StMungo tool [40], which is implemented as an ANTLR listener, and session typechecking is supported by modifying the original Ensemble typechecker to ensure that each communication action is permitted by the actor's declared session type.

Since EnsembleS builds directly on top of the original Ensemble implementation, it inherits Ensemble's runtime system. Performance results can be found in [26].

```
1  type Client is session(
2    connect RootServer;
3    RootRequest(DomainName) to RootServer;
4    choice at RootServer{
5      TLDResponse(ZoneServerAddress)
6          from RootServer;
7      disconnect RootServer;
8      rec Lookup {
9        connect ZoneServer;
10       ResolutionRequest(DomainName) to ZoneServer;
11       choice at ZoneServer {
12         PartialResolution(ZoneServerAddress)
13           from ZoneServer;
14         disconnect ZoneServer;
15         continue Lookup;
16       } or {
17         InvalidDomain(String) from ZoneServer;
18         disconnect ZoneServer;
19       } or {
20         ResolutionComplete(IPAddress)
21             from ZoneServer;
22         disconnect ZoneServer;
23       }
24     }
25   } or {
26     InvalidTLD(String) from RootServer;
27     disconnect RootServer;
28   }
29 )
```

Figure 7 EnsembleS DNS client `session` type.

4 Case study: DNS

To illustrate the use of session types for adaptive programming, we consider a real-world case study: the domain name system (DNS). DNS is a hierarchical, globally distributed translation system that converts an internet host name (domain name) into its corresponding numerical Internet Protocol (IP) address [43].

The process begins by transmitting a domain name to one of many well-known *root* servers. This server either rejects bad requests, or provides the information to contact a *zone* server. The zone server may know the IP address of the domain name; if not it refers the request to another zone server. This process continues until either the IP address is returned, or the name cannot be found.

To develop an adaptive DNS example, we assume no *a priori* information about server location, and instead use explicit discovery to find root and zone servers based on session types and server properties. We use an existing Scribble description of DNS as a starting point [21]. To illustrate adaptation we focus on the client who is querying DNS.

Fig. 7 shows the `session` type for the client actor which asks DNS to resolve a domain name. The client first asks for a root server (lines 2–3), and then either is informed that the request is invalid (lines 26–27) or recursively queries zone servers (lines 7–23) until the IP address is found (lines 20–22), or an error is reported (lines 17–18). Based on this `session`, StMungo generates EnsembleS types and interface definitions and a skeleton actor. Minimally completing the generated skeleton produces the code in Fig. 8.

In this example, discovery is used to locate the root server (lines 21–25, in Fig. 8) and the zone server (line 37). In each case, the `session` for the relevant server is provided to ensure that the discovered actor follows the expected protocol. When either server is located, the client `link`s with it (lines 26 and 39), enabling communication. When communication with the server is no longer required, the client `unlink`s explicitly (lines 33, 47, 51, 55, 62).

Although explicit discovery is used at the language level, there is nothing to prevent the implementation of discovery from caching the addresses of the root and zone servers. This does not affect the use of sessions in discovery or the safety they provide, as the type-based guarantees are still enforced. However, this would potentially improve performance of the system. Additionally, if a cached entry becomes stale, the full discovery process can again be used without code modification or degradation in trust.

A version of DNS which uses discovery allows the system to become more flexible and resilient to changing operational conditions, such as topology changes in the servers and their data. Session types ensure compatibility with the discovered actors.

```
type Iclient is interface(
 out{RootServer,string} RootServer_stringOut,
 in {RootServer,string} RootServer_stringIn,
 out{ZoneServer,string} ZoneServer_stringOut,
 in {ZoneServer,string} ZoneServer_stringIn,
 in {ZoneServer,choice_enum} ZoneServer_choiceIn,
 in {RootServer,choice_enum} RootServer_choiceIn)

type choice_enum is
 enum(TLDResponse,PartialResolution,
 InvalidDomain,ResolutionComplete,
 InvalidTLD)

query find_name(string n){ $name == n; }

actor c presents Iclient
 follows Client {
 dom_name = "nii.ac.jp";
 constructor() { }
 behaviour{
  rootQuery = find_name("jp");
  // Find Root Server
  root_s =
   discover(IServer, RootServer, rootQuery);
  // search until root_s non-empty
  link me with root_s[0];
  send domain_name on RootServer_stringOut;
  receive c_msg from RootServer_choiceIn;
  switch(c_msg){
   case TLDResponse:
    receive ZoneServerAddr_msg
      from RootServer_stringIn;
    unlink RootServer;
while(true) Lookup : {
 // Find ZoneServer
 zone_s =
    discover(IServer, ZoneServer,
      find_name(ZoneServerAddr_msg));
 link me with zone_s[0];
 // Ask ZoneServer
 send dom_name on ZoneServer_stringOut;
 receive c_msg2 from ZoneServer_choiceIn;
 switch(c_msg2){
  case PartialResolution:
   receive str_msg from ZoneServer_stringIn;
   ZoneServerAddr_msg := str_msg;
   unlink ZoneServer;
   continue Lookup;
  case InvalidDomain:
   receive str_msg from ZoneServer_stringIn;
   unlink ZoneServer;
   break;
  case ResolutionComplete:
   receive str_msg from ZoneServer_stringIn;
   unlink ZoneServer;
   break Lookup;
  }
  // keep looking
 }
case InvalidTLD:
  receive str_msg from RootServer_stringIn;
  unlink RootServer;
} } }
```

Figure 8 EnsembleS DNS client.

5 A Core Calculus for EnsembleS

In this section, we provide a formal characterisation of EnsembleS. In doing so, we show that our integration of adaptation with multiparty session types is safe, allowing adaptation while ruling out communication mismatches.

Relationship to implementation. Our core calculus aims to distil the essence of the interplay between adaptation and session-typed communication with explicit connection actions. Therefore, we concentrate on a functional core calculus rather than an imperative one: imperative variable binding serves only to clutter the formalism, and our fine-grain call-by-value representation can be thought of as an intermediate language.

Interfaces and unidirectional, simply-typed channels in EnsembleS are an implementation artifact: sending on a channel whose type changes is equivalent to sending on multiple channels with different types. Moreover, following theoretical accounts of multiparty session types [31, 14, 32], instead of having send and receive (resp. connect and accept) operations followed by branching (as done in Mungo and StMungo), we have unified **send** and **receive** constructs which communicate a label along with the message payload.

Since session typing is the interesting part of discovery, we omit properties and queries from the formalism; their inclusion is routine. Finally, we concentrate on *dynamic* topologies with explicit connection actions rather than static topologies since they are important for adaptation and more interesting technically.

Syntax of Types and Terms

Actor class names	u		
Actor definitions	D	::=	**actor** u **follows** S $\{M\}$
Roles	$\mathtt{p}, \mathtt{q}, \mathtt{s}, \mathtt{t}$		
Recursion Labels	l		
Behaviours	κ	::=	M | **stop**
Types	A, B	::=	$\mathsf{Pid}(S)$ | $\mathbf{1}$
Values	V, W	::=	x | $()$
Actions	L	::=	**return** V | **continue** l | **raise**
		|	**new** u | **self** | **replace** V **with** κ | **discover** S
		|	**connect** $\ell(V)$ **to** W **as** $\mathtt{p}$ | **accept from** $\mathtt{p}$ $\{\ell_i(x_i) \mapsto M_i\}_i$
		|	**send** $\ell(V)$ **to** $\mathtt{p}$ | **receive from** $\mathtt{p}$ $\{\ell_i(x_i) \mapsto M_i\}_i$
		|	**wait** $\mathtt{p}$ | **disconnect from** $\mathtt{p}$
Computations	M, N	::=	**let** $x \Leftarrow M$ **in** N | **try** L **catch** M | $l :: M$ | L

Syntax of Session Types

Session Actions	α, β	::=	$\mathtt{p}!\ell(A)$ | $\mathtt{p}!!\ell(A)$ | $\mathtt{p}?\ell(A)$ | $\mathtt{p}??\ell(A)$ | $\#{\uparrow}\mathtt{p}$
Session Types	S, T, U	::=	$\Sigma_{i\in I}(\alpha_i \,.\, S_i)$ | $\mu X.S$ | X | $\#{\downarrow}\mathtt{p}$ | end
Communication Actions	$\dagger$	::=	! | ?
Disconnection Actions	$\ddagger$	::=	$\#{\uparrow}$ | $\#{\downarrow}$

Figure 9 Syntax.

5.1 Syntax

Definitions. Figure 9 shows the syntax of Core EnsembleS terms and types. We let u range over actor class names and D range over definitions; each definition **actor** u **follows** S $\{M\}$ specifies the actor's class name, session type, and behaviour. Like class tables in Featherweight Java [36], we assume a fixed mapping from class names to definitions.

Values. Since our calculus is inherently effectful, we work in the setting of *fine-grain call-by-value* [41], where we have an explicit static stratification of values and computations and an explicit evaluation order similar to A-normal form [20]. Values V, W describe data that has been computed, and for the sake of simplicity, consist of variables and the unit value. Other base values (such as integers or booleans) can be encoded or added straightforwardly.

Computations. The **let** $x \Leftarrow M$ **in** N construct evaluates M, binding its result to x in N. The calculus supports exception handling over a single *action* L using **try** L **catch** M, where M is evaluated if L raises an exception, and labelled recursion using $l :: M$, stating that inside term M, a process can recurse to label l using **continue** l. *Actions* L denote the basic steps of a computation. The **return** V construct denotes a value.

Concurrency and adaptation constructs. The **new** u construct spawns a new actor of class u and returns its PID. The **self** construct returns the current actor's PID. An actor can replace the behaviour of itself or another actor V using **replace** V **with** κ. An actor can *discover* other actors following a session type S using the **discover** S construct, which returns the PID of the discovered actor.

Session communication constructs. An actor can connect to an actor W playing role $\mathtt{p}$ using **connect** $\ell(V)$ **to** W **as** $\mathtt{p}$, sending a message with label ℓ and payload V. An actor can accept a connection from another actor playing role $\mathtt{p}$ using **accept from** $\mathtt{p}$ $\{\ell_i(x_i) \mapsto M_i\}_i$, which allows an actor to receive a choice of messages; given a message with label ℓ_j, the payload is

bound to x_j in the continuation N_j. Once connected, an actor can communicate using the **send** and **receive** constructs. An actor can disconnect from p using **disconnect from** p, and await the disconnection of p using **wait** p.

Types. Types, ranged over by A, B, include the unit type **1** and process IDs $\mathsf{Pid}(S)$; the parameter S refers to the statically-known initial session type of the actor (i.e., the session type declared in the **follows** clause of a definition). Unlike in channel-based session-typed systems, process IDs themselves need not be linear: any number of actors can have a *reference* to another actor, but each actor may only be in a single session at a time. PIDs can be passed as payloads in session communications.

Session types. Session types are ranged over by S, T, U and follow the formulation of Hu & Yoshida [32]. A session type can be a choice of actions, written $\Sigma_{i\in I}(\alpha\,.\,S)$, a recursive session type $\mu X.S$ binding recursion variable X in continuation S, a recursion variable X, a disconnection action $\#{\downarrow}\mathtt{p}$, or the finished session end. The syntax of session types is more liberal than traditional "directed" presentations in order to allow output-directed choices to send or connect to different roles.

Session actions α involve sending (!), receiving (?), connecting (!!), or accepting (??) a message $\ell(A)$ with label ℓ and type A; or awaiting another participant's disconnection ($\#{\uparrow}$). As well as disallowing self-communication, following Hu & Yoshida [32], we require the following syntactic restrictions on session types:

▶ **Definition 1** (Syntactic validity). *A choice type $S = \Sigma_{i\in I}(\alpha_i\,.\,S_i)$ is* syntactically valid *if:*

1. *it is an output choice, i.e., each α_i is a send or connection action; or*
2. *it is a* directed *input choice, i.e., $S = \Sigma_{i\in I}(\mathtt{p}?\ell_i(A_i).S_i)$ or $S = \Sigma_{i\in I}(\mathtt{p}??\ell_i(A_i).S_i)$; or*
3. *the choice consists of single wait action $\#{\uparrow}\mathtt{p}\,.\,S$.*

In the remainder of the paper, we assume that all session types are syntactically valid.

Session correlation. The most general form of explicit connection actions allows a participant to leave and re-join a session, or accept connections from multiple different participants. Such generality comes at a cost, since care must be taken to ensure that the *same* participant plays the role throughout the session.

To address this *session correlation* issue, Hu & Yoshida [32] propose two solutions: either augment global types with type-level assertions and check conformance dynamically, or adopt a lightweight syntactic restriction which requires that each local type must contain at most a single accept action as its top-level construct. We opt for the latter, enforcing the constraint as part of our safety property (§5.4.2), and by requiring that $\#{\downarrow}\mathtt{p}$ does not have a continuation. (Note that the behaviour will repeat, so p will be able to accept again after disconnecting). As Hu & Yoshida [32] show, this design still supports the most common use cases of explicit connection actions.

Global types. Traditional MPST works [31, 14] use *global types* to describe the interactions between participants at a global level, which are then projected into *local types*; projectability ensures safety and deadlock-freedom.

Since we are using explicit connection actions, traditional approaches are insufficiently flexible as they do not account for certain roles being present in certain branches but not others. Following [53] and subsequently non-classical MPSTs [54], we instead formulate our typing rules and safety properties using collections of local types.

It is, however, still convenient to write a global type and have local types computed programatically. Global types are defined as follows:

$$\begin{array}{lrcl} \text{Global Actions} & \pi & ::= & \mathtt{p} \to \mathtt{q} : \ell(A) \mid \mathtt{p} \twoheadrightarrow \mathtt{q} : \ell(A) \mid \mathtt{p}\#\mathtt{q} \\ \text{Global Types} & G & ::= & \Sigma_{i \in I}(\pi_i \,.\, G_i) \mid \mu X.G \mid X \mid \mathsf{end} \end{array}$$

Global actions π describe interactions between participants: $\mathtt{p} \to \mathtt{q} : \ell(A)$ states that role $\mathtt{p}$ sends a message with label ℓ and payload type A to $\mathtt{q}$. Similarly, $\mathtt{p} \twoheadrightarrow \mathtt{q} : \ell(A)$ states that $\mathtt{p}$ connects to $\mathtt{q}$ by sending a message with label ℓ and payload type A. The disconnection action $\mathtt{p}\#\mathtt{q}$ states that role $\mathtt{p}$ disconnects from role $\mathtt{q}$.

We can write the `OnlineStore` example from § 2 as follows:

$$\begin{array}{l} \mathtt{Customer} \twoheadrightarrow \mathtt{Store} : \mathit{login}(\mathsf{String}) \,.\, \mu\mathsf{Browse} \,. \\ \quad \mathtt{Customer} \to \mathtt{Store} : \mathit{item}(\mathsf{String}) \,.\, \mathtt{Store} \to \mathtt{Customer} : \mathit{price}(\mathsf{Int}) \,.\, \mathsf{Browse} \\ \quad + \\ \quad \mathtt{Customer} \to \mathtt{Store} : \mathit{address}(\mathsf{String}) \,.\, \mathtt{Store} \twoheadrightarrow \mathtt{Courier} : \mathit{deliver}(\mathsf{String}) \,. \\ \quad \mathtt{Courier} \to \mathtt{Store} : \mathit{ref}(\mathsf{Int}) \,.\, \mathtt{Courier}\#\mathtt{Store} \,.\, \mathtt{Store} \to \mathtt{Customer} : \mathit{ref}(\mathsf{Int}) \,. \\ \quad \mathtt{Store}\#\mathtt{Customer} \,.\, \mathsf{end} \\ \quad + \\ \quad \mathtt{Customer} \to \mathtt{Store} : \mathit{quit}(\mathbf{1}) \,.\, \mathtt{Store}\#\mathtt{Customer} \,.\, \mathsf{end} \end{array}$$

Although projectability in our setting does not necessarily guarantee safety and deadlock-freedom, we show a projection algorithm, adapted from that of Hu & Yoshida [32], in the extended version. The resulting local types can then be checked for safety (§5.4.2).

Protocols and Programs. Terms do not live in isolation; they refer to a set of *protocols*, and evaluate in the context of an actor. A *protocol* maps role names to local session types.

▶ **Definition 2** (Protocol). *A* protocol *is a set* $\{\mathtt{p}_i : S_i\}_i$ *mapping role names to session types.*

As an example, consider the protocol for the online shop example:

$$\left\{ \begin{array}{l} \mathtt{Customer} : \mathtt{Store}!!\mathit{login}(\mathsf{String}) \,.\, \mu\mathsf{Browse} \,. \\ \quad \mathtt{Store}!\mathit{item}(\mathsf{String}) \,.\, \mathtt{Store}?\mathit{price}(\mathsf{Int}) \,.\, \mathsf{Browse} \\ \quad + \;\; \mathtt{Store}!\mathit{address}(\mathsf{String}) \,.\, \mathtt{Store}?\mathit{ref}(\mathsf{Int}) \,.\, \#{\uparrow}\mathtt{Store} \,.\, \mathsf{end} \\ \quad + \;\; \mathtt{Store}!\mathit{quit}(\mathbf{1}) \,.\, \#{\uparrow}\mathtt{Store} \,.\, \mathsf{end}, \\ \\ \mathtt{Store} : \mathtt{Customer}??\mathit{login}(\mathsf{String}) \,.\, \mu\mathsf{Browse} \,. \\ \quad \mathtt{Customer}?\mathit{item}(\mathsf{String}) \,.\, \mathtt{Customer}!\mathit{price}(\mathsf{Int}) \,.\, \mathsf{Browse} \\ \quad + \;\; \mathtt{Customer}?\mathit{address}(\mathsf{String}) \,.\, \mathtt{Courier}!!\mathit{deliver}(\mathsf{String}) \,.\, \mathtt{Courier}?\mathit{ref}(\mathsf{Int}) \,. \\ \quad \#{\uparrow}\mathtt{Courier} \,.\, \mathtt{Customer}!\mathit{ref}(\mathsf{Int}) \,.\, \#{\downarrow}\mathtt{Customer} \\ \quad + \;\; \mathtt{Customer}?\mathit{quit}(\mathbf{1}) \,.\, \#{\downarrow}\mathtt{Customer}, \\ \\ \mathtt{Courier} : \mathtt{Store}??\mathit{deliver}(\mathsf{String}) \,.\, \mathtt{Store}!\mathit{ref}(\mathsf{Int}) \,.\, \#{\downarrow}\mathtt{Store} \end{array} \right\}$$

We can now consider an implementation of a `Store` actor, which uses discovery to find a courier. We write **receive** $\ell(x)$ **from** $\mathtt{p}$; M and **accept** $\ell(x)$ **from** $\mathtt{p}$; M as syntactic sugar for **receive from** $\mathtt{p}$ $\{\ell(x) \mapsto M\}$ and **accept from** $\mathtt{p}$ $\{\ell(x) \mapsto M\}$ respectively, and write $M; N$ as syntactic sugar for $\mathbf{let} x \Leftarrow M \,\mathbf{in}\, N$ for a fresh variable x. We assume the existence of a function lookupPrice, and define CourierType as $\mathtt{Store}??\mathit{deliver}(\mathsf{String}) \,.\, \mathtt{Store}!\mathit{ref}(\mathsf{Int}) \,.\, \#{\downarrow}\mathtt{Store}$.

```
actor Store follows ty(Store) {
  accept login(credentials) from Customer;
  Browse ::
      receive from Customer {
          item(name) ↦
              send price(lookupPrice(name)) to Customer;
              continue Browse
          address(addr) ↦
              let pid ⇐ discover CourierType in
              connect deliver(addr) to pid as Courier;
              receive ref(r) from Courier;
              wait Courier;
              send ref(r) to Customer;
              disconnect from Customer
          quit(()) ↦ disconnect from Customer
      }
}
```

A *program* consists of actor definitions, protocol definitions, and the "boot" clause to be run in order to set up initial actor communication.

▶ **Definition 3** (Program). *An EnsembleS program is a 3-tuple* $(\overrightarrow{D}, \overrightarrow{P}, M)$ *of a set of definitions, protocols, and an initial term to be evaluated.*

In the context of a program, we write $\mathsf{ty}(\mathtt{p})$ to refer to the session type associated with role p as defined by the set of protocols. Given an actor definition **actor** u **follows** S $\{M\}$, we define $\mathsf{sessionType}(u) = S$ and $\mathsf{behaviour}(u) = M$.

5.2 Typing rules

Figures 10 and 11 show the typing rules for EnsembleS. Value typing, with judgement $\Gamma \vdash V{:}A$, states that under environment Γ, value V has type A. Judgement $\vdash D$ states that an actor definition **actor** u **follows** S $\{M\}$ is well-typed if its body is typable under, and fully consumes, its statically-defined session type S. The behaviour typing judgement $\{S\}\ \Gamma \vdash \kappa$ states that given static session type S, behaviour κ is well-typed under Γ. Specifically, **stop** is always well-typed, and M is well-typed if it is typable under and fully consumes S.

5.2.1 Term typing

The typing judgement for terms $\{T\}\ \Gamma \mid S \triangleright M{:}A \triangleleft S'$ reads "in an actor following T, under typing environment Γ and with current session type S, term M has type A and updates the session type to S'". Note that the term typing judgement, reminiscent of parameterised monads [3], contains a session precondition S and may perform some session communication actions to arrive at postcondition S'.

Functional rules. Rule T-Let is a sequencing operation: given a construct **let** $x \Leftarrow M$ **in** N where M has pre-condition S and post-condition S', and where N has pre-condition S' and post-condition S'', the overall construct has pre-condition S and post-condition S''.

Following Kouzapas *et al.* [40], we formalise recursion through annotated expressions: term $l :: M$ states that M is an expression which can loop to l by evaluating **continue** l. We take an equi-recursive view of session types, identifying recursive sessions with their unfolding ($\mu X.S = S\{\mu X.S/X\}$), and assume that recursion is guarded. Rule T-Rec extends the typing environment with a recursion label defined at the current session type. Rule T-Continue ensures that the pre-condition must match the label stored in the environment, but has arbitrary type and any post-condition since the return type and post-condition depend on the enclosing loop's base case.

Definition typing $\boxed{\vdash D}$ **Value typing** $\boxed{\Gamma \vdash V{:}A}$ **Behaviour typing** $\boxed{\{S\}\ \Gamma \vdash \kappa}$

$$\text{T-Def}\ \frac{\{S\}\ \cdot\ \mid\ S \triangleright M{:}A \triangleleft \mathsf{end}}{\vdash \mathbf{actor}\ u\ \mathbf{follows}\ S\ \{M\}} \qquad \text{T-Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x{:}A} \qquad \text{T-Unit}\ \frac{}{\Gamma \vdash ():\mathbf{1}} \qquad \text{T-Stop}\ \frac{}{\{S\}\ \Gamma \vdash \mathbf{stop}} \qquad \text{T-Body}\ \frac{\{S\}\ \Gamma\ \mid\ S \triangleright M{:}A \triangleleft \mathsf{end}}{\{S\}\ \Gamma \vdash M}$$

Typing rules for computations $\boxed{\{T\}\ \Gamma\ \mid\ S \triangleright M{:}A \triangleleft S'}$

Functional Rules

$$\text{T-Let}\ \frac{\{T\}\ \Gamma\ \mid\ S \triangleright M{:}A \triangleleft S' \qquad \{T\}\ \Gamma, x : A\ \mid\ S' \triangleright N{:}B \triangleleft S''}{\{T\}\ \Gamma\ \mid\ S \triangleright \mathbf{let}\ x \Leftarrow M\ \mathbf{in}\ N{:}B \triangleleft S''} \qquad \text{T-Return}\ \frac{\Gamma \vdash V{:}A}{\{T\}\ \Gamma\ \mid\ S \triangleright \mathbf{return}\ V{:}A \triangleleft S}$$

$$\text{T-Rec}\ \frac{\{T\}\ \Gamma, l : S\ \mid\ S \triangleright M{:}A \triangleleft S'}{\{T\}\ \Gamma\ \mid\ S \triangleright l :: M{:}A \triangleleft S'} \qquad \text{T-Continue}\ \frac{}{\{T\}\ \Gamma, l : S\ \mid\ S \triangleright \mathbf{continue}\ l{:}A \triangleleft S'}$$

Actor / Adaptation Rules

$$\text{T-New}\ \frac{\mathsf{sessionType}(u) = U}{\{T\}\ \Gamma\ \mid\ S \triangleright \mathbf{new}\ u{:}\mathsf{Pid}(U) \triangleleft S} \qquad \text{T-Self}\ \frac{}{\{T\}\ \Gamma\ \mid\ S \triangleright \mathbf{self}{:}\mathsf{Pid}(T) \triangleleft S} \qquad \text{T-Discover}\ \frac{}{\{T\}\ \Gamma\ \mid\ S \triangleright \mathbf{discover}\ U{:}\mathsf{Pid}(U) \triangleleft S}$$

$$\text{T-Replace}\ \frac{\Gamma \vdash V{:}\mathsf{Pid}(U) \qquad \{U\}\ \Gamma \vdash \kappa}{\{T\}\ \Gamma\ \mid\ S \triangleright \mathbf{replace}\ V\ \mathbf{with}\ \kappa{:}\mathbf{1} \triangleleft S}$$

Figure 10 Typing rules (1).

Actor and adaptation rules. Rule T-New states that creating an actor of class u returns a PID parameterised by the session type declared in the class of u. Rule T-Self retrieves a PID for the current actor, parameterised by the statically-defined session type of the local actor (i.e., the T in the judgement $\{T\}\ \Gamma\ \mid\ S \triangleright M{:}A \triangleleft S'$). Rule T-Discover states **discover** U returns a PID of type $\mathsf{Pid}(U)$. Finally, given a behaviour κ typable under a static session type U, and a process ID with the matching static type $\mathsf{Pid}(U)$, T-Replace allows replacement, and returns the unit type.

Exception handling rules. Figure 11 shows the rules for exception handling and session communication. T-Raise denotes raising an exception; since it does not return, it can have an arbitrary return type and postcondition. Rule T-Try types an exception handler **try** L **catch** M which acts over a single action L. If L raises an exception, then M is evaluated instead. Since L only scopes over a single action, the **try** and **catch** clauses have the same pre- and post-conditions to allow the action to be retried if necessary.

▶ Remark 4. Following Mostrous & Vasconcelos [45], our **try** L **catch** M construct scopes over a *single* action and is discarded afterwards. We opt for this simple approach since in our setting exceptions are a means to an end, but (at the cost of a more involved type system) we could potentially scope over multiple actions as long as the handler is compatible with all potential exit conditions [23]. We leave a thorough exploration to future work.

Session communication rules. Rule T-Conn types a term **connect** $\ell_j(V)$ **to** W **as** $\mathtt{p}_j$. Given the precondition is a choice type containing a branch $\mathtt{p}!!\ell_j(A_j)\,.\,S'_j$, and the remote actor reference is W of type $\mathsf{Pid}(S)$, the rule ensures that S is compatible with the type of $\mathtt{p}_j$, and ensures that the label and payload are compatible with the session type. The session type is then advanced to S'_j. Rule T-Send follows the same pattern.

Exception handling rules

$$\textsc{T-Raise}\ \frac{}{\{T\}\ \Gamma\ \mid\ S \triangleright \mathbf{raise}{:}A \triangleleft S'} \qquad \textsc{T-Try}\ \frac{\{T\}\ \Gamma\ \mid\ S \triangleright L{:}A \triangleleft S' \qquad \{T\}\ \Gamma\ \mid\ S \triangleright M{:}A \triangleleft S'}{\{T\}\ \Gamma\ \mid\ S \triangleright \mathbf{try}\ L\ \mathbf{catch}\ M{:}A \triangleleft S'}$$

Session communication rules

$$\textsc{T-Conn}\ \frac{\mathtt{p}_j!!\ell_j(A_j) \in \{\alpha_i\}_{i\in I} \qquad \Gamma \vdash V{:}A_j \qquad \Gamma \vdash W{:}\mathsf{Pid}(T) \qquad T = \mathsf{ty}(\mathtt{p}_j)}{\{T\}\ \Gamma\ \mid\ \Sigma_{i\in I}(\alpha_i\,.\,S_i) \triangleright \mathbf{connect}\ \ell_j(V)\ \mathbf{to}\ W\ \mathbf{as}\ \mathtt{p}_j{:}\mathbf{1} \triangleleft S'_j}$$

$$\textsc{T-Send}\ \frac{\mathtt{p}_j!\ell_j(A_j) \in \{\alpha_i\}_{i\in I} \qquad \Gamma \vdash V{:}A_j}{\{T\}\ \Gamma\ \mid\ \Sigma_{i\in I}(\alpha_i\,.\,S_i) \triangleright \mathbf{send}\ \ell_j(V)\ \mathbf{to}\ \mathtt{p}_j{:}\mathbf{1} \triangleleft S'_j}$$

$$\textsc{T-Accept}\ \frac{(\{T\}\ \Gamma, x_i : B_i\ \mid\ S_i \triangleright M_i{:}A \triangleleft S)_{i\in I}}{\{T\}\ \Gamma\ \mid\ \Sigma_{i\in I}(\mathtt{q}??\ell_i(B_i)\,.\,S_i) \triangleright \mathbf{accept\ from}\ \mathtt{q}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}{:}A \triangleleft S}$$

$$\textsc{T-Recv}\ \frac{(\{T\}\ \Gamma, x_i : B_i\ \mid\ S_i \triangleright M_i{:}A \triangleleft S)_{i\in I}}{\{T\}\ \Gamma\ \mid\ \Sigma_{i\in I}(\mathtt{q}?\ell_i(B_i)\,.\,S_i) \triangleright \mathbf{receive\ from}\ \mathtt{q}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}{:}A \triangleleft S} \qquad \textsc{T-Wait}\ \frac{}{\{T\}\ \Gamma\ \mid\ \#{\uparrow}\mathtt{q}\,.\,S \triangleright \mathbf{wait}\ \mathtt{q}{:}\mathbf{1} \triangleleft S}$$

$$\textsc{T-Disconn}\ \frac{}{\{T\}\ \Gamma\ \mid\ \#{\downarrow}\mathtt{q} \triangleright \mathbf{disconnect\ from}\ \mathtt{q}{:}\mathbf{1} \triangleleft \mathsf{end}}$$

Figure 11 Typing rules (2).

Given a session type $\Sigma_{i\in I}(\mathtt{p}??x_i(A_i))\,.\,S_i$, rule T-Accept types term **accept from** p $\{\ell_i(x_i) \mapsto M_i\}_{i\in I}$, enabling an actor to accept connections with messages ℓ_i, binding the payload x_i in each continuation M_i. Like **case** expressions in functional languages, each continuation must be typable under an environment extended with $x_i : A_i$, under session type S_i, and each branch must have same result type and postcondition. Rule T-Recv is similar.

Rule T-Wait handles waiting for a participant p to disconnect from a session, requiring a pre-condition of $\#{\uparrow}\mathtt{p}\,.\,S$, returning the unit type and advancing the session type to S. Rule T-Disconnect is similar and advances the session type to end.

5.3 Operational semantics

We describe the semantics of EnsembleS via a deterministic reduction relation on terms, and a nondeterministic reduction relation on configurations.

5.3.1 Runtime syntax

Figure 12 shows the runtime syntax and the first part of the reduction rules for EnsembleS.

Whereas static syntax and typing rules describe code that a user would write, runtime syntax arises during evaluation. We introduce two types of runtime name: s ranges over *session names*, which are created when a process initiates a session, and a ranges over *actor names*, which uniquely identify each actor once it has been spawned by **new**.

Configurations. Configurations, ranged over by $\mathcal{C}, \mathcal{D}, \mathcal{E}$, represent the concurrent fragment of the language. Like in the π-calculus [42], name restrictions $(\nu n)\mathcal{C}$ bind name n in $\mathcal{C}$, $\mathcal{C} \parallel \mathcal{D}$ denotes $\mathcal{C}$ and $\mathcal{D}$ running in parallel, and the $\mathbf{0}$ configuration denotes the inactive process.

Runtime syntax

Names	n	::=	$a \mid s$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E}$	::=	$(\nu n)\mathcal{C} \mid \mathcal{C} \parallel \mathcal{D} \mid \langle a, M, \sigma, \kappa\rangle \mid \lightning\, s[\mathtt{p}] \mid \mathbf{0}$
Connection state	σ	::=	$\bot \mid s[\mathtt{p}]\langle\tilde{\mathtt{q}}\rangle$
Runtime environments	Δ	::=	$\cdot \mid \Delta, a : S \mid \Delta, s[\mathtt{p}]\langle\tilde{\mathtt{q}}\rangle{:}S$
Evaluation contexts	E	::=	$F \mid \mathbf{let}\ x \Leftarrow E\ \mathbf{in}\ M$
Top-level contexts	F	::=	$[\,] \mid \mathbf{try}\ [\,]\ \mathbf{catch}\ M$
Pure contexts	E_{P}	::=	$[\,] \mid \mathbf{let}\ x \Leftarrow E_{\mathsf{P}}\ \mathbf{in}\ M$

Term reduction $\boxed{M \longrightarrow_{\mathsf{M}} N}$

E-Let	$\mathbf{let}\ x \Leftarrow \mathbf{return}\ V\ \mathbf{in}\ M$	$\longrightarrow_{\mathsf{M}}$	$M\{V/x\}$
E-TryReturn	$\mathbf{try}\ \mathbf{return}\ V\ \mathbf{catch}\ M$	$\longrightarrow_{\mathsf{M}}$	$\mathbf{return}\ V$
E-TryRaise	$\mathbf{try}\ \mathbf{raise}\ \mathbf{catch}\ M$	$\longrightarrow_{\mathsf{M}}$	M
E-Rec	$l :: M$	$\longrightarrow_{\mathsf{M}}$	$M\{l :: M/\mathbf{continue}\ l\}$
E-LiftM	$E[M]$	$\longrightarrow_{\mathsf{M}}$	$E[N]$ if $M \longrightarrow_{\mathsf{M}} N$

Configuration reduction (1) $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

Actor / adaptation rules

E-Loop
$$\frac{}{\langle a, \mathbf{return}\ V, \bot, M\rangle \longrightarrow \langle a, M, \bot, M\rangle}$$

E-New
$$\frac{b \text{ is fresh} \qquad \mathsf{behaviour}(u) = M}{\langle a, E[\mathbf{new}\ u], \sigma, \kappa\rangle \longrightarrow (\nu b)(\langle a, E[\mathbf{return}\ b], \sigma, \kappa\rangle \parallel \langle b, M, \bot, M\rangle)}$$

E-Replace
$$\frac{}{\langle a, E[\mathbf{replace}\ b\ \mathbf{with}\ \kappa'], \sigma_1, \kappa_1\rangle \parallel \langle b, N, \sigma_2, \kappa_2\rangle \longrightarrow \langle a, E[\mathbf{return}\ ()], \sigma_1, \kappa_1\rangle \parallel \langle b, N, \sigma_2, \kappa'\rangle}$$

E-ReplaceSelf
$$\frac{}{\langle a, E[\mathbf{replace}\ a\ \mathbf{with}\ \kappa'], \sigma, \kappa\rangle \longrightarrow \langle a, E[\mathbf{return}\ ()], \sigma, \kappa'\rangle}$$

E-Discover
$$\frac{\mathsf{sessionType}(b) = S \qquad \neg((N = \mathbf{return}\ V \vee N = \mathbf{raise}) \wedge \kappa_2 = \mathbf{stop})}{\langle a, E[\mathbf{discover}\ S], \sigma_1, \kappa_1\rangle \parallel \langle b, E'[N], \sigma_2, \kappa_2\rangle \longrightarrow \langle a, E[\mathbf{return}\ b], \sigma_1, \kappa_1\rangle \parallel \langle b, E'[N], \sigma_2, \kappa_2\rangle}$$

E-Self
$$\frac{}{\langle a, E[\mathbf{self}], \sigma, \kappa\rangle \longrightarrow \langle a, E[\mathbf{return}\ a], \sigma, \kappa\rangle}$$

Figure 12 Operational semantics (1).

Actors are represented at runtime as a 4-tuple $\langle a, M, \sigma, \kappa\rangle$, where a is the actor's runtime name; M is the term currently evaluating; σ is the connection state; and κ is the actor's current behaviour. A connection state is either *disconnected*, written $\bot$, or playing role $\mathtt{p}$ in session s and connected to roles $\tilde{\mathtt{q}}$, written $s[\mathtt{p}]\langle\tilde{\mathtt{q}}\rangle$.

Inspired by Mostrous & Vasconcelos [45] and Fowler *et al.* [22], a *zapper thread* $\lightning\, s[\mathtt{p}]$ indicates that participant $\mathtt{p}$ in session s cannot be used for future communications, for example due to the actor playing the role crashing due to an unhandled exception.

To run a program, we place it in an *initial configuration.* of the form $(\nu a)(\langle a, M, \bot, \mathbf{stop}\rangle)$.

Runtime typing environments. Whereas Γ is an unrestricted typing environment used for typing values and configurations, we introduce Δ as a linear runtime environment. Runtime environments can contain entries of type $a : S$, stating that actor a has *statically-defined* session type S, and entries of type $s[\mathtt{p}]\langle\tilde{\mathtt{q}}\rangle{:}S$, stating that in session s, role $\mathtt{p}$ is connected to roles $\tilde{\mathtt{q}}$ and *currently* has session type S.

Configuration reduction (2) $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

Session reduction rules

E-ConnInit

$$\frac{j \in I}{\langle a, E[F[\textbf{connect}\ \ell_j(V)\ \textbf{to}\ b\ \textbf{as}\ \mathtt{q}]], \bot, \kappa_1\rangle \parallel \langle b, E'[F'[\textbf{accept from}\ \mathtt{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}]], \bot, \kappa_2\rangle \longrightarrow \langle \nu s)(\langle a, E[\textbf{return}\ ()], s[\mathtt{p}]\langle \mathtt{q}\rangle, \kappa_1\rangle \parallel \langle b, E'[M_j\{V/x_j\}], s[\mathtt{q}]\langle \mathtt{p}\rangle, \kappa_2\rangle)}$$

E-Conn

$$\frac{\mathtt{q} \notin \tilde{\mathtt{r}}}{\langle a, E[F[\textbf{connect}\ \ell_j(V)\ \textbf{to}\ b\ \textbf{as}\ \mathtt{q}]], s[\mathtt{p}]\langle \tilde{\mathtt{r}}\rangle, \kappa_1\rangle \parallel \langle b, E'[F'[\textbf{accept from}\ \mathtt{p}\ \{\ell_i(x_i) \mapsto N_i\}_{i\in I}]], \bot, \kappa_2\rangle \longrightarrow \langle a, E[\textbf{return}\ ()], s[\mathtt{p}]\langle \tilde{\mathtt{r}}, \mathtt{q}\rangle, \kappa_1\rangle \parallel \langle b, E'[N_j\{V/x_j\}], s[\mathtt{q}]\langle \mathtt{p}\rangle, \kappa_2\rangle}$$

E-ConnFail

$$\frac{((N = \textbf{return}\ V \vee N = E_{\mathsf{P}}[\textbf{raise}]) \wedge \kappa_2 = \textbf{stop}) \vee \sigma_2 \neq \bot}{\langle a, E[\textbf{connect}\ \ell_j(V)\ \textbf{to}\ b\ \textbf{as}\ \mathtt{q}], \sigma_1, \kappa_1\rangle \parallel \langle b, N, \sigma_2, \kappa_2\rangle \longrightarrow \langle a, E[\textbf{raise}], \sigma_1, \kappa_1\rangle \parallel \langle b, N, \sigma_2, \kappa_2\rangle}$$

E-Disconn

$$\frac{}{\langle a, E[F[\textbf{wait}\ \mathtt{q}]], s[\mathtt{p}]\langle \tilde{\mathtt{r}}, \mathtt{q}\rangle, \kappa_1\rangle \parallel \langle b, E'[F'[\textbf{disconnect from}\ \mathtt{p}]], s[\mathtt{q}]\langle \mathtt{p}\rangle, \kappa_2\rangle \longrightarrow \langle a, E[\textbf{return}\ ()], s[\mathtt{p}]\langle \tilde{\mathtt{r}}\rangle, \kappa_1\rangle \parallel \langle b, E'[\textbf{return}\ ()], \bot, \kappa_2\rangle}$$

E-Comm

$$\frac{j \in I \qquad \mathtt{q} \in \tilde{\mathtt{r}} \qquad \mathtt{p} \in \tilde{\mathtt{s}}}{\langle a, E[F[\textbf{send}\ \ell_j(V)\ \textbf{to}\ \mathtt{q}]], s[\mathtt{p}]\langle \tilde{\mathtt{r}}\rangle, \kappa_1\rangle \parallel \langle b, E'[F'[\textbf{receive from}\ \mathtt{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}]], s[\mathtt{q}]\langle \tilde{\mathtt{s}}\rangle, \kappa_2\rangle \longrightarrow \langle a, E[\textbf{return}\ ()], s[\mathtt{p}]\langle \tilde{\mathtt{r}}\rangle, \kappa_1\rangle \parallel \langle b, E'[M_j\{V/x_j\}], s[\mathtt{q}]\langle \tilde{\mathtt{s}}\rangle, \kappa_2\rangle}$$

E-Complete

$$\frac{}{(\nu s)(\langle a, \textbf{return}\ V, s[\mathtt{p}]\langle \emptyset\rangle, \kappa\rangle) \longrightarrow \langle a, \textbf{return}\ V, \bot, \kappa\rangle}$$

Figure 13 Operational semantics (2).

Evaluation contexts. Due to our fine-grain call-by-value presentation, evaluation contexts E allow nesting only in the immediate subterm of a **let** expression. The top-level frame F can either be a hole, or a single, top-level exception handler. Pure contexts E_{P} do not include exception handling frames.

5.3.2 Reduction rules

Term reduction $\longrightarrow_{\mathsf{M}}$ is standard β-reduction, save for E-TryRaise which evaluates the failure continuation in the case of an exception. We consider four subcategories of configuration reduction rules: actor and adaptation rules; session communication rules; exception handling rules; and administrative rules.

Actor / adaptation rules. Given a fully-evaluated actor, E-Loop runs the term specified by the actor's behaviour. Rule E-New allows actor a to spawn a new actor of class u by creating a fresh runtime actor name b and a new actor process of the form $\langle b, M, \bot, M\rangle$ where M is the behaviour specified by u, returning the process ID b. Rules E-Replace and E-ReplaceSelf handle replacement by changing the behaviour of an actor, returning the unit value to the caller. Rule E-Discover returns the process ID of an actor b if it has the desired static session type S. Rule E-Self returns the PID of the local actor.

Session communication rules. An actor begins a session by connecting to another actor while disconnected; such a case is handled by rule E-ConnInit. Suppose we have a disconnected actor a evaluating a connection statement **connect** $\ell_j(V)$ **to** b **as** $\mathtt{p}$, evaluating in

Configuration reduction (3) $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

Exception handling rules

$$\text{E-CommRaise}\ \frac{\mathsf{subj}(M) = \mathtt{q}}{\langle a, E[M], s[\mathtt{p}]\langle \tilde{\mathtt{r}} \rangle, \kappa \rangle \parallel \lightning\, s[\mathtt{q}] \longrightarrow \langle a, E[\mathbf{raise}], s[\mathtt{p}]\langle \tilde{\mathtt{r}} \rangle, \kappa \rangle \parallel \lightning\, s[\mathtt{q}]} \qquad \text{E-FailS}\ \frac{}{\langle a, E_{\mathsf{P}}[\mathbf{raise}], s[\mathtt{p}]\langle \tilde{\mathtt{r}} \rangle, \kappa \rangle \longrightarrow \langle a, \mathbf{raise}, \bot, \kappa \rangle \parallel \lightning\, s[\mathtt{p}]} \qquad \text{E-FailLoop}\ \frac{}{\langle a, E_{\mathsf{P}}[\mathbf{raise}], \bot, M \rangle \longrightarrow \langle a, M, \bot, M \rangle}$$

Administrative rules

$$\text{E-LiftM}\ \frac{M \longrightarrow_{\mathsf{M}} M'}{\langle a, E[M], \sigma, \kappa \rangle \longrightarrow \langle a, E[M'], \sigma, \kappa \rangle} \qquad \text{E-Equiv}\ \frac{\mathcal{C} \equiv \mathcal{C}' \quad \mathcal{C}' \longrightarrow \mathcal{D}' \quad \mathcal{D}' \equiv \mathcal{D}}{\mathcal{C} \longrightarrow \mathcal{D}} \qquad \text{E-Par}\ \frac{\mathcal{C} \longrightarrow \mathcal{C}'}{\mathcal{C} \parallel \mathcal{D} \longrightarrow \mathcal{C}' \parallel \mathcal{D}} \qquad \text{E-Nu}\ \frac{\mathcal{C} \longrightarrow \mathcal{D}}{(\nu n)\mathcal{C} \longrightarrow (\nu n)\mathcal{D}}$$

Configuration equivalence $\boxed{\mathcal{C} \equiv \mathcal{D}}$

$$\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \qquad \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} \qquad (\nu n_1)(\nu n_2)\mathcal{C} \equiv (\nu n_2)(\nu n_1)\mathcal{C}$$

$$\mathcal{C} \parallel (\nu n)\mathcal{D} \equiv (\nu n)(\mathcal{C} \parallel \mathcal{D}) \quad \text{if } n \notin \mathsf{fn}(\mathcal{C}) \qquad (\nu s)(\lightning\, s[\mathtt{p}_1] \parallel \cdots \parallel \lightning\, s[\mathtt{p}_n]) \parallel \mathcal{C} \equiv \mathcal{C} \qquad \mathcal{C} \parallel \mathbf{0} \equiv \mathcal{C}$$

Figure 14 Operational semantics (3).

parallel with a disconnected actor b evaluating an accept statement $\mathbf{accept\ from}\ \mathtt{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i \in I}$. Rule E-ConnInit returns the unit value to actor a; creates a fresh session name restriction s, sets the connection state of a to $s[\mathtt{p}]\langle \mathtt{q} \rangle$ and of b to $s[\mathtt{q}]\langle \mathtt{p} \rangle$; accepting actor b then evaluates continuation M_j with V substituted for x_j. Since exception handlers only scope over a single communication action, the top-level frames F, F' in each actor are discarded if the communication succeeds. Rule E-Conn handles the case where the connecting actor is already part of a session and behaves similarly to E-ConnInit, without creating a new session name restriction. A connection can fail if an actor attempts to connect to another actor which is terminated or is already involved in a session; in these cases, E-ConnFail raises an exception in the connecting actor.

Rule E-Disconn handles the case where an actor b leaves a session, synchronising with an actor a. In this case, the unit value is returned to both callers, and the connection state of b is set to $\bot$. Rule E-Comm handles session communication when two participants are already connected to the same session, and is similar to E-Conn. Rule E-Complete garbage collects a session after it has completed and sets the initiator's connection state to $\bot$.

Exception handling rules. Exception handling rules allow safe session communication in the presence of exceptions. Rule E-CommRaise states that if an actor is attempting to communicate with a role no longer present due to an exception, then an exception should be raised. We write $\mathsf{subj}(E[M]) = \mathtt{p}$ if $M \in \{\mathbf{send}\ \ell(V)\ \mathbf{to}\ \mathtt{p}, \mathbf{receive\ from}\ \mathtt{p}\ \{\ell_i(x_i) \mapsto N_i\}_i, \mathbf{wait}\ \mathtt{p}, \mathbf{disconnect\ from}\ \mathtt{p}\}$. Rule E-FailS states that if a connected actor encounters an unhandled exception, then a zapper thread will be generated for the current role, the actor will become disconnected, and the current evaluation context will be discarded. Rule E-FailLoop restarts an actor encountering an unhandled exception.

Administrative rules. The remaining rules are administrative: E-LiftM allows term reduction inside an actor; E-Equiv allows reduction modulo structural congruence; E-Par allows reduction under parallel composition; and E-Nu allows reduction under name restrictions.

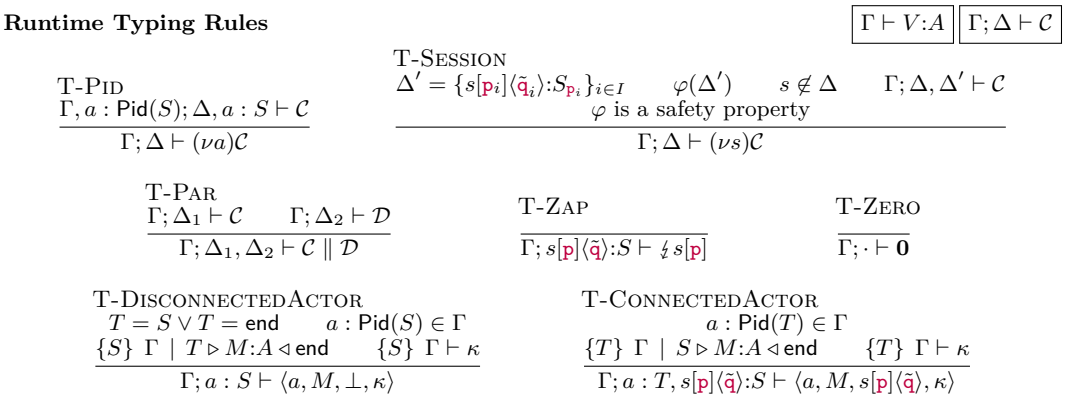


Figure 15 Runtime typing rules.

Configuration equivalence. Reduction includes configuration equivalence $\equiv$, defined as the smallest congruence relation satisfying the axioms in Figure 14. The equivalence rules extend the usual π-calculus structural congruence rules with a "garbage collection" equivalence, which allows us to discard a session where all participants have exited due to an error.

5.4 Metatheory

We now turn our attention to showing that session typing allows runtime adaptation and discovery while precluding communication mismatches and deadlocks.

5.4.1 Runtime typing

To reason about the metatheory, we introduce typing rules for configurations (Fig. 15): the judgement $\Gamma; \Delta \vdash \mathcal{C}$ states that configuration $\mathcal{C}$ is well-typed under term typing environment Γ and runtime typing environment Δ.

Rule T-Pid types actor name restriction $(\nu a)\mathcal{C}$ by adding a PID into the term environment, and extending the runtime typing environment $a : S$; the linearity of the runtime typing environment therefore means that the system must contain precisely one actor with name a.

Session name restrictions $(\nu s)\mathcal{C}$ are typed by T-Session. We follow the formulation of Scalas & Yoshida [54] which types multiparty sessions using a parametric safety property φ; we discuss safety properties in more depth in Section 5.4.2. Let Δ' be a runtime typing environment containing only mappings of the form $s[\mathtt{p}_i]\langle\tilde{\mathtt{q}}_i\rangle{:}S_i$. Assuming Δ does not contain any mappings involving session s and Δ' satisfies φ, the rule states that $\mathcal{C}$ is typable under typing environment Γ and runtime typing environment Δ, Δ'. It is sometimes convenient to annotate session ν-binders with their environment, e.g., $(\nu s : \Delta')\mathcal{C}$.

Rule T-Par types each subconfiguration of a parallel composition by splitting the linear runtime environment. Rule T-Zap types a zapper thread $\lightning\, s[\mathtt{p}]$, assuming the runtime environment contains an entry $s[\mathtt{p}]\langle\tilde{\mathtt{q}}\rangle{:}S$ for any session type S.

Finally, rules T-DisconnectedActor and T-ConnectedActor type disconnected and connected actor configurations respectively. Given an actor with name a and static session type T, both rules require that the typing environment contains $a : \mathsf{Pid}(T)$ and runtime environment contains $a : T$. Both rules require that the current session type is fully consumed by the currently-evaluating term and that the actor's behaviour should be typable under T. Rule T-DisconnectedActor requires that the currently-evaluating term must be typable under either T or end, whereas to type a connection state of $s[\mathtt{p}]\langle\tilde{\mathtt{q}}\rangle$ and current session type S, T-ConnectedActor requires an entry $s[\mathtt{p}]\langle\tilde{\mathtt{q}}\rangle{:}S$ in the runtime environment.

Labels

$$\begin{array}{lllll} \text{Labels} & \gamma & ::= & s{:}\mathtt{p}\dagger\mathtt{q}{::}\ell(A) \mid s{:}\mathtt{p} \twoheadrightarrow \mathtt{q}{::}\ell \mid s{:}\mathtt{p}\ddagger\mathtt{q} \\ \text{Synchronisation labels} & \rho & ::= & s{:}\mathtt{p},\mathtt{q}{::}\ell \mid s{:}\mathtt{p} \twoheadrightarrow \mathtt{q}{::}\ell \mid s{:}\mathtt{p}\#\mathtt{q} \end{array}$$

Reduction on runtime typing environments

Local Reduction $\boxed{\Delta \xrightarrow{\gamma} \Delta'}$

ET-Conn
$$\frac{\exists j \in I.\alpha_j = \mathtt{q}!!\ell_j(A_j) \quad \mathsf{ty}(\mathtt{q}) = \Sigma_{k\in K}(\mathtt{p}??\ell_k(B_k)\,.\,T_k) \qquad j \in K \qquad A_j = B_j}{s[\mathtt{p}]\langle\tilde{\mathtt{r}}\rangle{:}\Sigma_{i\in I}(\alpha_i\,.\,S_i) \xrightarrow{s:\mathtt{p}\twoheadrightarrow\mathtt{q}::\ell_j} s[\mathtt{p}]\langle\tilde{\mathtt{r}},\mathtt{q}\rangle{:}S_j, s[\mathtt{q}]\langle\mathtt{p}\rangle{:}T_j}$$

ET-Act
$$\frac{\exists j \in I.\alpha_j = \mathtt{q}\dagger\ell_j(A_j) \qquad \mathtt{q} \in \tilde{\mathtt{r}}}{s[\mathtt{p}]\langle\tilde{\mathtt{r}}\rangle{:}\Sigma_{i\in I}(\alpha_i\,.\,S_i) \xrightarrow{s:\mathtt{p}\dagger\mathtt{q}::\ell_j(A_j)} s[\mathtt{p}]\langle\tilde{\mathtt{r}}\rangle{:}S_j}$$

ET-Wait
$$\frac{}{s[\mathtt{p}]\langle\tilde{\mathtt{r}},\mathtt{q}\rangle{:}\#\uparrow\mathtt{q}\,.\,S \xrightarrow{s:\mathtt{p}\#\uparrow\mathtt{q}} s[\mathtt{p}]\langle\tilde{\mathtt{r}}\rangle{:}S}$$

ET-Disconn
$$\frac{}{s[\mathtt{p}]\langle\mathtt{q}\rangle{:}\#\downarrow\mathtt{q} \xrightarrow{s:\mathtt{p}\#\downarrow\mathtt{q}} \cdot}$$

ET-Rec
$$\frac{\Delta, s[\mathtt{p}]\langle\tilde{\mathtt{q}}\rangle{:}S\{\mu X.S/X\} \xrightarrow{\gamma} \Delta'}{\Delta, s[\mathtt{p}]\langle\tilde{\mathtt{q}}\rangle{:}\mu X.S \xrightarrow{\gamma} \Delta'}$$

ET-Cong1
$$\frac{\Delta \xrightarrow{\gamma} \Delta'}{\Delta, s[\mathtt{p}]\langle\tilde{\mathtt{q}}\rangle{:}S \xrightarrow{\gamma} \Delta', s[\mathtt{p}]\langle\tilde{\mathtt{q}}\rangle{:}S}$$

ET-Cong2
$$\frac{\Delta \xrightarrow{\gamma} \Delta'}{\Delta, a{:}S \xrightarrow{\gamma} \Delta', a{:}S}$$

Synchronisation $\boxed{\Delta \xRightarrow{\rho} \Delta'}$

ET-ConnSync
$$\frac{\Delta \xrightarrow{s:\mathtt{p}\twoheadrightarrow\mathtt{q}::\ell} \Delta'}{\Delta \xRightarrow{s:\mathtt{p}\twoheadrightarrow\mathtt{q}::\ell} \Delta'}$$

ET-Comm
$$\frac{\Delta_1 \xrightarrow{s:\mathtt{p}!\mathtt{q}::\ell(A)} \Delta_1' \qquad \Delta_2 \xrightarrow{s:\mathtt{q}?\mathtt{p}::\ell(A)} \Delta_2'}{\Delta_1, \Delta_2 \xRightarrow{s:\mathtt{p},\mathtt{q}::\ell} \Delta_1', \Delta_2'}$$

ET-Disconn
$$\frac{\Delta_1 \xrightarrow{s:\mathtt{p}\#\uparrow\mathtt{q}} \Delta_1' \qquad \Delta_2 \xrightarrow{s:\mathtt{q}\#\downarrow\mathtt{p}} \Delta_2'}{\Delta_1, \Delta_2 \xRightarrow{s:\mathtt{p}\#\mathtt{q}} \Delta_1', \Delta_2'}$$

Figure 16 Labelled transition system for runtime typing environments.

5.4.2 Preservation

We now prove that reduction preserves typability and thus that actors only perform communication actions specified in their session types. Due to our use of explicit connection actions, classical MPST approaches are too limited for our purposes. Our approach, following that of Scalas & Yoshida [54], is to introduce a labelled transition system (LTS) on local types, and specify a generic safety property based around local type reduction. The property can then refined; in our case, we will later specialise the property in order to prove progress.

Reduction on runtime typing environments. Figure 16 shows the LTS on runtime typing environments. There are two judgements: $\Delta \xrightarrow{\gamma} \Delta'$, which handles reduction of individual local types, and a *synchronisation* judgement $\Delta \xRightarrow{\rho} \Delta'$.

Rule ET-Conn handles the reduction of role $\mathtt{p}$, where the choice session type contains a connection action $\mathtt{q}!!\ell_j(A_j)\,.\,S_j'$. If $\mathtt{q}$ has a statically-defined session type $\Sigma_{k\in K}(\mathtt{p}??\ell_k(B_k)\,.\,T_k)$ which can accept ℓ_j from from $\mathtt{p}$, and the payload types match, reduction advances $\mathtt{p}$'s session type, adds $\mathtt{q}$ to $\mathtt{p}$'s connected role set, and introduces an entry for $\mathtt{q}$ into the environment. The reduction emits a label $s{:}\mathtt{p} \twoheadrightarrow \mathtt{q}{::}\ell_j$.

Given a role $\mathtt{p}$ connected to $\mathtt{q}$ with a session choice containing a send or receive action $\mathtt{q}\dagger\ell_j(A)\,.\,S_j'$, rule ET-Act will emit a label $s{:}\mathtt{p}\dagger\mathtt{q}{::}\ell_j(A_j)$ and advance the session type of $\mathtt{p}$.

Rule ET-Wait handles the reduction of $\#\uparrow\mathtt{q}\,.\,S$ actions, $s[\mathtt{p}]\langle\tilde{\mathtt{r}},\mathtt{q}\rangle{:}\#\uparrow\mathtt{q}\,.\,S$, where $\mathtt{p}$ waits for $\mathtt{q}$ to disconnect: the reduction emits label $\mathtt{p}{:}\mathtt{q}\#\uparrow$ and removes $\mathtt{q}$ from $\mathtt{p}$'s connected roles.

Similarly, rule ET-DISCONN handles disconnection, by emitting label $\mathtt{p}{:}\mathtt{q}\#{\downarrow}$ and removing the entry from the environment. ET-REC handles recursive types, and the ET-CONG rules handle reduction of sub-environments.

Rule ET-CONNSYNC states that connection is a synchronisation action, and rules ET-COMM and ET-DISCONN handle synchronisation between dual actions in sub-environments, emitting synchronisation labels $s{:}\mathtt{p}, \mathtt{q}{::}\ell$ and $s{:}\mathtt{p}\#\mathtt{q}$ respectively. We omit the congruence rules for synchronisation actions. We say that a runtime environment *reduces*, written $\Delta \Longrightarrow$, if there exists some Δ' such that $\Delta \Longrightarrow \Delta'$.

Safety. A *safety property* describes a set of invariants on typing environments which allow us to prove preservation. Since the type system is parametric in the given safety property, we can tweak the property to permit or rule out different typing environments satisfying particular behavioural properties; however, we need only prove type preservation once, using the weakest safety property. Our safety property is different to the safety property described by Scalas & Yoshida [54] in order to account for explicit connection actions.

▶ **Definition 5** (Safety Property). *φ is a* safety property *of runtime typing contexts Δ if:*

1. $\varphi(\Delta, s[\mathtt{p}]\langle\tilde{\mathtt{r}}\rangle{:}\Sigma_{i\in I}(\alpha_i \,.\, S_i), s[\mathtt{q}]\langle\tilde{\mathtt{s}}\rangle{:}\Sigma_{j\in J}(\mathtt{p}?\ell_j(B_j) \,.\, T_j))$ *implies that if* $\mathtt{q}!\ell_k(A_k) \in \{\alpha_i\}_{i\in I}$, *then* $k \in J$, $\mathtt{q} \in \tilde{\mathtt{r}}$, $\mathtt{p} \in \tilde{\mathtt{s}}$, *and* $A_k = B_k$.
2. $\varphi(\Delta, s[\mathtt{p}]\langle\tilde{\mathtt{r}}\rangle{:}\Sigma_{i\in I}(\alpha_i \,.\, S_i))$ *implies that if* $\alpha_i = \mathtt{q}!!\ell_j(A_j) \in \{\alpha_i\}_{i\in I}$, *then* $\mathtt{q} \notin \tilde{\mathtt{r}}$, $s[\mathtt{q}]\langle\tilde{\mathtt{s}}\rangle \notin \mathit{dom}(\Delta)$, *and* $\mathit{ty}(\mathtt{q}) = \Sigma_{k\in K}(\mathtt{p}??\ell_k(B_k) \,.\, T_k)$ *with* $j \in K$ *and* $A_j = B_j$.
3. $\varphi(\Delta, s[\mathtt{p}]\langle\tilde{\mathtt{q}}\rangle{:}\mu X.S)$ *implies* $\varphi(\Delta, s[\mathtt{p}]\langle\tilde{\mathtt{q}}\rangle{:}S\{\mu X.S/X\})$
4. $\varphi(\Delta)$ *and* $\Delta \Longrightarrow \Delta'$ *implies* $\varphi(\Delta')$

A runtime typing environment is safe, *written* $\mathit{safe}(\Delta)$, *if* $\varphi(\Delta)$ *for a safety property* φ.

Clause (1) ensures that communication actions between participants are compatible: if $\mathtt{p}$ is sending a message with label ℓ and payload type A to $\mathtt{q}$, and $\mathtt{q}$ is receiving from $\mathtt{p}$, then the two roles must be connected, and $\mathtt{q}$ must be able to receive ℓ with a matching payload.

Clause (2) states that if $\mathtt{p}$ is connecting to a role $\mathtt{q}$ with label ℓ, then $\mathtt{q}$ should not already be involved in the session, and should be able to accept from $\mathtt{p}$ on message label ℓ with a compatible payload type. The requirement that $\mathtt{q}$ is not already involved in the session rules out the *correlation* errors described in Section 5.2.1. Clause (3) handles recursion, and clause (4) requires that safety is preserved under environment reduction.

Concretising the safety property. In order to deduce that a runtime typing environment Δ is safe, we define $\varphi(\Delta) = \{\Delta' \mid \Delta \Longrightarrow^* \Delta'\}$ and verify that φ is a safety property by ensuring that it satisfies all clauses in Definition 5.

Properties on protocols and programs. It is useful to distinguish active and inactive session types, depending on whether their associated role is currently involved in a session, and identify the initiator of a session.

▶ **Definition 6** (Active and Inactive Session Types). *A session type S is* inactive, *written* $\mathit{inactive}(S)$, *if* $S = \mathit{end}$ *or* $S = \Sigma_{i\in I}(\mathtt{p}??\ell_i(A_i) \,.\, S_i)$. *Otherwise, S is* active, *written* $\mathit{active}(S)$.

▶ **Definition 7** (Initiator, unique initiator). *Given a protocol P, a role $\mathtt{p} : S_\mathtt{p} \in P$ is an* initiator *if* $S_\mathtt{p} = \Sigma_{i\in I}(\alpha_i \,.\, S_i)$, *and each α_i is a connection action* $\mathtt{q}!!\ell_i(A_i)$. *Role $\mathtt{p}$ is a* unique initiator *of P if* $\mathit{inactive}(S_\mathtt{q})$ *for all* $\mathtt{q} \in P \setminus \{\mathtt{p} : S_\mathtt{p}\}$.

A protocol is *well-formed* if it is safe and has a unique initiator.

▶ **Definition 8** (Well-formed protocol). *A protocol* $P = \{\mathtt{p}_i : S_i\}_{i \in I}$ *is* well-formed *if it has a unique initiator* $\mathtt{q}$ *of type* S *and* $\mathsf{safe}(s[\mathtt{q}]\langle\emptyset\rangle{:}S)$ *for any* s.

By way of example, the online shopping protocol is well-formed: `Customer` is the protocol's unique initiator, and it is straightforward to verify that $\mathsf{safe}(s[\mathtt{Customer}]\langle\emptyset\rangle{:}\,\mathsf{ty}(\mathtt{Customer}))$.

▶ **Definition 9** (Well-formed program). *A program* $(\overrightarrow{D}, \overrightarrow{P}, M)$ *is* well-formed *if:*

1. *For each actor definition* $D = \textbf{\textit{actor}}\ u\ \textbf{\textit{follows}}\ S\ \{N\} \in \overrightarrow{D}$, *there exists some role* $\mathtt{p} \in \overrightarrow{P}$ *such that* $\mathsf{ty}(\mathtt{p}) = S$, *and* $\{S\} \cdot \mid S \triangleright N{:}A \triangleleft \mathsf{end}$
2. *Each protocol* $P \in \overrightarrow{P}$ *is well-formed and has a distinct set of roles*
3. *The "boot clause"* M *is typable under the empty typing environment and does not perform any communication actions:* $\{\mathsf{end}\} \cdot \mid \mathsf{end} \triangleright M{:}A \triangleleft \mathsf{end}$

When discussing the metatheory, we only consider configurations defined with respect to a well-formed program. Specifically, we henceforth assume that each actor definition in the system follows a session type matched by a role in a given protocol, assume each role belongs to a single protocol, and assume that all protocols are well-formed.

Given a safe runtime environment, configuration reduction preserves typability; details can be found in the extended version. We write $\mathcal{R}^?$ for the reflexive closure of a relation $\mathcal{R}$.

▶ **Theorem 10** (Preservation (Configurations)). *Suppose* $\Gamma; \Delta \vdash \mathcal{C}$ *with* $\mathsf{safe}(\Delta)$ *and where* $\mathcal{C}$ *is defined wrt. a well-formed program. If* $\mathcal{C} \longrightarrow \mathcal{C}'$, *then there exists some* Δ' *such that* $\Delta \Longrightarrow^? \Delta'$ *and* $\Gamma; \Delta' \vdash \mathcal{C}'$.

Preservation shows that each actor conforms to its session type, and that communication never introduces unsoundness due to mismatching payload types.

5.4.3 Progress

We now show a progress property, which shows that given *protocols* which satisfy a progress property, EnsembleS *configurations* do not get stuck due to deadlocks.

A *final* runtime typing environment contains a single, disconnected role of type end, reflecting the intuition that all roles will eventually disconnect from a protocol initiator.

▶ **Definition 11** (Final environment). *An environment* Δ *is* final, *written* $\mathsf{end}(\Delta)$, $\Delta = \{s[\mathtt{p}]\langle\emptyset\rangle{:}\mathsf{end}\}$ *for some* s *and* $\mathtt{p}$.

So far, we have considered *safe* protocols, which ensure the absence of communication mismatches. We say that an environment *satisfies progress* if each active role can eventually perform an action, each potential send is eventually matched by a receive, and non-reducing environments are final. Let $\mathsf{roles}(\rho)$ denote the roles referenced in a synchronisation label (i.e., $\mathsf{roles}(\rho) = \{\mathtt{p}, \mathtt{q}\}$ for $\rho \in \{s{:}\mathtt{p} \twoheadrightarrow \mathtt{q}{::}\ell, s{:}\mathtt{p}, \mathtt{q}{::}\ell, s{:}\mathtt{p}\#\mathtt{q}\}$).

▶ **Definition 12** (Progress (Runtime typing environments)). *A runtime typing environment* Δ satisfies progress, *written* $\mathsf{prog}(\Delta)$, *if:*

- *(Role progress) for each* $s[\mathtt{p}_i]\langle\tilde{\mathtt{q}}_i\rangle{:}S_i \in \Delta$ *s.t.* $\mathsf{active}(S_i)$, $\Delta \Longrightarrow^* \Delta' \overset{\rho}{\Longrightarrow}$ *with* $\mathtt{p} \in \mathsf{roles}(\rho)$
- *(Eventual comm.) if* $\Delta \Longrightarrow^* \Delta' \xrightarrow{s:\mathtt{p}!\mathtt{q}::\ell(A)}$, *then* $\Delta' \overset{\overrightarrow{\rho}}{\Longrightarrow} \Delta'' \xrightarrow{s:\mathtt{q}?\mathtt{p}::\ell(A)}$, *with* $\mathtt{p} \notin \mathsf{roles}(\overrightarrow{\rho})$
- *(Correct termination)* $\Delta \Longrightarrow^* \Delta' \not\Longrightarrow$ *implies* $\mathsf{end}(\Delta)$

The online shopping example satisfies progress, since all roles will always eventually be able to fire an action once connected, and since all roles disconnect, the non-reducing final environment will be of the form $s[\mathtt{Customer}]\langle\emptyset\rangle{:}\mathsf{end}$.

▶ **Definition 13** (Progress (Programs)). *A well-formed program* $(\overrightarrow{D}, \overrightarrow{P}, M)$ satisfies progress *if each* $P \in \overrightarrow{P}$ *has a unique initiator* $\mathtt{q}$ *of type* S *and* $\mathsf{prog}(s[\mathtt{q}]\langle\emptyset\rangle{:}S)$ *for any* s.

A *configuration context* $\mathcal{G}$ is the context $\mathcal{G} ::= [\,] \mid (\nu s)\mathcal{G} \mid \mathcal{G} \parallel \mathcal{C}$. A *session* consists of a session name restriction and all connected actors and zapper threads. A well-typed configuration can be written as a sequence of sessions, followed by all disconnected actors.

▶ **Definition 14** (Session). *A configuration is a* session $\mathcal{S}$ *if it can be written:*
$(\nu s)(\langle a_1, M_1, s[\mathtt{p}_1]\langle\tilde{\mathtt{q}}_1\rangle, \kappa_1\rangle \parallel \cdots \parallel \langle a_m, M_m, s[\mathtt{p}_m]\langle\tilde{\mathtt{q}}_m\rangle, \kappa_m\rangle \parallel \lightning\, s[\mathtt{p}_{m+1}] \parallel \cdots \parallel \lightning\, s[\mathtt{p}_n])$

An actor is *terminated* if it has reduced to a value or has an unhandled exception, and has the behaviour **stop**. An unmatched discover occurs when no other actors match a given session type. An actor is *accepting* if it is ready to accept a connection.

▶ **Definition 15** (Terminated actor, unmatched discover, accepting actor).

- *An actor* $\langle a, M, \sigma, \kappa\rangle$ *is* terminated *if* $M = \mathbf{return}\, V$ *or* $M = E_P[\mathbf{raise}]$, *and* $\kappa = \mathbf{stop}$.
- *An actor* $\langle a, E[\mathbf{discover}\ S], \sigma, \kappa\rangle$ *which is a subconfiguration of* $\mathcal{C}$ *has an* unmatched discover *if no other non-terminated actor in* $\mathcal{C}$ *has session type* S.
- *An actor* $\langle a, M, \sigma, \kappa\rangle$ *is* accepting *if* $M = E[\mathbf{accept\ from}\ \mathtt{p}\ \{\ell_j(x_j) \mapsto N_j\}_j]$ *for some evaluation context* E *and role* $\mathtt{p}$.

Unhandled exceptions will propagate through a session, progressively cancelling all roles. A *failed session* consists of only zapper threads.

▶ **Definition 16** (Failed session). *We say that a session* $\mathcal{S}$ *is a* failed session, *written* $\mathsf{failed}(\mathcal{S})$, *if* $\mathcal{S} \equiv (\nu s)(\lightning\, s[\mathtt{p}_1] \parallel \cdots \parallel \lightning\, s[\mathtt{p}_n])$.

The key *session progress* lemma establishes the reducibility of each session which does not contain an unmatched discover and is typable under a reducible runtime typing environment.

▶ **Lemma 17** (Session Progress). *If* $\cdot;\cdot \vdash \mathcal{C}$ *where* $\mathcal{C}$ *does not contain an unmatched discover,* $\mathcal{C} \equiv \mathcal{G}[\mathcal{S}]$ *and* $\mathcal{S} = (\nu s : \Delta)\mathcal{D}$ *with* $\mathsf{prog}(\Delta)$, *and* $\mathcal{S}$ *is not a failed session, then* $\mathcal{C} \longrightarrow$.

There are several steps to proving Lemma 17 (see the extended version). First, we introduce *exception-aware* reduction on runtime typing environments, which explicitly accounts for zapper threads at the type level, and show that exception-aware environments threads retain safety and progress. Second, we introduce *flattenings*, which show that runtime typing environments containing only unary output choices can type configurations blocked on communication actions, and that flattenings preserve environment reducibility. Finally, we show that configurations typable under flat, reducible typing environments can reduce.

We can now show our second main result: in the absence of unmatched discovers, a configuration can either reduce, or it consists only of accepting and terminating actors.

▶ **Theorem 18** (Progress). *Suppose* $\cdot;\cdot \vdash \mathcal{C}$ *where* $\mathcal{C}$ *is defined wrt. a well-formed program which satisfies progress, and* $\mathsf{prog}(\Delta)$ *for each* $(\nu s : \Delta)\mathcal{C}'$ *in* $\mathcal{C}$. *If* $\mathcal{C}$ *does not contain an unmatched discover, either* $\exists \mathcal{D}$ *such that* $\mathcal{C} \longrightarrow \mathcal{D}$, *or* $\mathcal{C} \equiv \mathbf{0}$, *or* $\mathcal{C} \equiv (\nu b_1 \cdots \nu b_n)(\langle b_1, N_1, \bot, \kappa_1\rangle \parallel \cdots \parallel \langle b_n, N_n, \bot, \kappa_n\rangle)$ *where each* b_i *is terminated or accepting.*

The proof eliminates all failed sessions by the structural congruence rules; shows that the presence of sessions implies reducibility (Lem. 17); and reasons about disconnected actors.

In addition to each actor conforming to its session type (Thm. 10), Theorem 18 guarantees that the system does not deadlock. It follows that session types ensure safe communication.

Theorem 18 assumes the absence of unmatched discovers. This is not a significant limitation in practice, however, as unmatched discovers can be mitigated with timeouts, where a timeout would trigger an exception.

6 Related Work

Behavioural typing for actors. Mostrous & Vasconcelos [46] present the first theoretical account of session types in an actor language; their work effectively overlays a channel-based session typing discipline on mailboxes using Erlang's reference generation capabilities.

Neykova & Yoshida [49] use MPSTs to specify communication in an actor system, implemented in Python. Fowler [21] implements similar ideas in Erlang, with extensions to allow subsessions [17] and failure handling. Neykova & Yoshida [48] later improve the recovery mechanism of Erlang by using MPSTs to calculate a minimal set of affected roles. The above works check multiparty session typing dynamically. We are first to both formalise and implement static multiparty session type checking for an actor language.

Active objects (AOs) [15] are actor-like concurrent objects where results of asynchronous method invocations are returned through futures. Bagherzadeh & Rajan [4] study *order types* for an AO calculus, which characterise causality and statically rule out data races. In contrast to MPSTs, order types work bottom-up through type inference. Kamburjan *et al.* [39] apply an MPST-like system to Core ABS [38], a core AO calculus; they establish soundness via a translation to register automata rather than via an operational semantics.

de'Liguoro & Padovani [16] introduce *mailbox types*, a type system for first-class, unordered mailboxes. Their calculus generalises the actor model, since each process can be associated with more than one mailbox. Their type discipline allows multiple writers and a single reader for each mailbox, and ensures conformance, deadlock-freedom, and for many programs, junk-freedom. Our approach is based on MPSTs and is more process-centric.

Non-classical multiparty session types. MPSTs were introduced by Honda *et al.* [31]. *Classical* MPST theories are grounded in binary duality: safety follows as a consequence of *consistency* (pointwise binary duality of interactions between participants), and deadlock-freedom follows from projectability from a global type.

Unfortunately, classical MPSTs are restrictive: there are many protocols which are intuitively safe but not consistent. Scalas & Yoshda [54] introduced the first *non-classical* multiparty session calculus. Instead of ensuring safety using binary duality, they define an LTS on local types and *safety property* suitable for proving type preservation; since the type system is *parametric* in the safety property (inspired by Igarashi & Kobayashi [35] in the π-calculus), the property can be customised in order to guarantee different properties such as deadlock-freedom or liveness. Hu & Yoshida [32] formalise MPSTs with explicit connection actions via an LTS on types rather than providing a concrete language design or calculus; in our setting, a calculus is vital in order to account for the impact of adaptation constructs. A key contribution of our work is the use of non-classical MPSTs to prove preservation and progress properties for a calculus incorporating MPSTs with explicit connection actions.

Adaptation. None of the above work considers adaptation. The literature on formal studies of adaptation is mainly based on process calculi, without programming language design or implementation. Bravetti *et al.* [9] develop a process calculus that allows parts of a process to be dynamically replaced with new definitions. Their later work [8] uses temporal logic rather than types to verify adaptive processes. Di Giusto and Pérez [19] define a session type system for the same process calculus, and prove that adaptation does not disrupt active sessions. Later, Di Giusto and Pérez [18] use an event-based approach so that adaptation can depend on the state of a session protocol. Anderson and Rathke [2] develop an MPST-like calculus and study dynamic software update providing guarantees of communication safety and liveness. Differently from our work, they do not consider runtime discovery of software components and do not provide an implementation.

Coppo *et al.* [13] consider self-adaptation, in which a system reconfigures itself rather than receiving external updates. They define an MPST calculus with self-adaptation and prove type safety. Castellani *et al.* [11] extend [13] to also guarantee properties of secure information flow, but neither of these works have been implemented. Dalla Preda *et al.* [51] develop the AIOCJ system based on choreographic programming for runtime updates. Their work is implemented in the Jolie language [44], but they do not consider runtime discovery.

In this work we focus on correct communication in the absence of adversaries, and do not consider security. The literature on security and behavioural types is surveyed by Bartoletti *et al.* [5] and could provide a basis for future work on security properties.

7 Conclusion and Future Work

Modern computing increasingly requires software components to *adapt* to their environment, by *discovering*, *replacing*, and *communicating* with other components which may not be part of the system's original design. Unfortunately, up until now, existing programming languages have lacked the ability to support adaptation both *safely* and *statically*. We therefore asked:

> *Can a programming language support static (compile-time) verification of safe runtime dynamic self-adaptation, i.e., discovery, replacement and communication?*

We have answered this question in the affirmative by introducing EnsembleS, an actor-based language supporting adaptation, which uses multiparty session types to guarantee communication safety, using explicit connection actions to invite discovered actors into a session. We have demonstrated the safety of our system by proving type soundness theorems which state that each actor follows its session type, and that communication does not introduce deadlocks. Our formalism makes essential use of *non-classical* MPSTs.

Future work. Each actor only takes part in a single session. Unlike dynamically-checked implementations of session typing for actors [47, 21], this means that a message received by an actor in one session cannot trigger an interaction in another (e.g., the Warehouse example in [47]). A key focus for future work will be to allow actors to partake in multiple sessions.

EnsembleS discovery and replacement requires type equality. We expect we could relax this constraint to subtyping [53] or perhaps bisimilarity on local types to increase expressiveness.

In order to avoid session correlation errors, we require that each role includes at most a single top-level **accept** construct (c.f. [32]). It would be interesting to investigate the more general setting, which would likely require dependent types.

References

1 Gul A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990.

2 Gabrielle Anderson and Julian Rathke. Dynamic software update for message passing programs. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, volume 7705 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2012. doi:10.1007/978-3-642-35182-2_15.

3 Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335–376, 2009.

4 Mehdi Bagherzadeh and Hridesh Rajan. Order types: static reasoning about message races in asynchronous message passing concurrency. In *AGERE!@SPLASH*, pages 21–30. ACM, 2017.

5 Massimo Bartoletti, Ilaria Castellani, Pierre-Malo Deniélou, Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Jovanka Pantovic, Jorge A. Pérez, Peter Thiemann, Bernardo Toninho, and Hugo Torres Vieira. Combining behavioural types with security analysis. *Journal of Logical and Algebraic Methods in Programming*, 84(6):763–780, 2015. doi:10.1016/j.jlamp.2015.09.003.

6 J. Baumann, F. Hohl, K. Rothermel, and M. Straßer. *MOLE — Concepts of Mobile Agent System*, page 535–554. ACM Press/Addison-Wesley Publishing Co., USA, 1999.

7 G. Bouabene, C. Jelger, C. Tschudin, S. Schmid, A. Keller, and M. May. The autonomic network architecture (ANA). *IEEE Journal on Selected Areas in Communications*, 28(1):4–14, 2010. doi:10.1109/JSAC.2010.100102.

8 Mario Bravetti, Cinzia Di Giusto, Jorge A. Pérez, and Gianluigi Zavattaro. Adaptable processes. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/LMCS-8(4:13)2012.

9 Mario Bravetti, Cinzia Di Giusto, Jorge A. Pérez, and Gianluigi Zavattaro. Towards the verification of adaptable processes. In *Proceedings (Part I) of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 7609 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2012. doi:10.1007/978-3-642-34026-0_20.

10 Callum Cameron, Paul Harvey, and Joseph Sventek. A virtual machine for the Insense language. In *Proceedings of the International Conference on Mobile Wireless Middleware, Operating Systems and Applications (Mobilware)*, pages 1–10. IEEE, 2013. doi:10.1109/Mobilware.2013.17.

11 Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Jorge A. Pérez. Self-adaptation and secure information flow in multiparty communications. *Formal Aspects of Computing*, 28(4):669–696, 2016. doi:10.1007/s00165-016-0381-3.

12 Francesco Cesarini and Steve Vinoski. *Designing for Scalability with Erlang/OTP: Implement Robust, Fault-Tolerant Systems*. O'Reilly Media, Inc., 1st edition, 2016.

13 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Self-adaptive multiparty sessions. *Service Oriented Computing and Applications*, 9(3-4):249–268, 2015. doi:10.1007/s11761-014-0171-9.

14 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016.

15 Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2007.

16 Ugo de'Liguoro and Luca Padovani. Mailbox types for unordered interactions. In *ECOOP*, volume 109 of *LIPIcs*, pages 15:1–15:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

17 Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR*, volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012.

18 Cinzia Di Giusto and Jorge A. Pérez. Disciplined structured communications with disciplined runtime adaptation. *Science of Computer Programming*, 97:235–265, 2015. doi:10.1016/j.scico.2014.04.017.

19 Cinzia Di Giusto and Jorge A. Pérez. An event-based approach to runtime adaptation in communication-centric systems. In *Proceedings of the 11th and 12th International Workshops on Web Services, Formal Methods and Behavioural Types (WS-FM 2014, WS-FM/BEAT 2015)*, Lecture Notes in Computer Science, pages 67–85. Springer, 2015. doi:10.1007/978-3-319-33612-1_5.

20 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247. ACM, 1993.

21 Simon Fowler. An Erlang implementation of multiparty session actors. In *Proceedings of the 9th Interaction and Concurrency Experience (ICE)*, volume 223 of *Electronic Proceedings in Theoretical Computer Science*, pages 36–50. Open Publishing Association, 2016. doi:10.4204/EPTCS.223.3.

22 Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, 2019.

23 Colin S. Gordon. Lifting sequential effects to control operators. In *ECOOP*, volume 166 of *LIPIcs*, pages 23:1–23:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

24 T. Gu, H. K. Pung, and D. Q. Zhang. Toward an OSGi-based infrastructure for context-aware applications. *IEEE Pervasive Computing*, 3(4):66–74, 2004. doi:10.1109/MPRV.2004.19.

25 Paul Harvey. *A linguistic approach to concurrent, distributed, and adaptive programming across heterogeneous platforms*. PhD thesis, School of Computing Science, University of Glasgow, 2015. URL: http://theses.gla.ac.uk/6749/.

26 Paul Harvey and Joseph Sventek. Adaptable actors: just what the world needs. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS)*, pages 22–28. ACM, 2017. doi:10.1145/3144555.3144559.

27 Richard Hayton, Michael Bursell, Douglas I. Donaldson, W. Harwood, and Andrew Herbert. Mobile Java objects. *Distributed Syst. Eng.*, 6(1):51, 1999. doi:10.1088/0967-1846/6/1/306.

28 Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL: http://dl.acm.org/citation.cfm?id=1624775.1624804.

29 Kohei Honda. Types for dyadic interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

30 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

31 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, volume 43, pages 273–284. ACM, 2008. doi:10.1145/1328897.1328472.

32 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE)*, Lecture Notes in Computer Science, pages 116–133. Springer, 2017. doi:10.1007/978-3-662-54494-5_7.

33 Danny Hughes, Klaas Thoelen, Wouter Horré, Nelson Matthys, Javier Del Cid, Sam Michiels, Christophe Huygens, and Wouter Joosen. LooCI: a loosely-coupled component infrastructure for networked embedded systems. In *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia (MoMM)*, pages 195–203. ACM, 2009. doi:10.1145/1821748.1821787.

34 Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 81–94. ACM, 2004. doi:10.1145/1031495.1031506.

35 Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.

36 Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

37 Antonio J Jara, Pedro Martinez-Julia, and Antonio Skarmeta. Light-weight multicast DNS and DNS-SD (lmDNS-SD): IPv6-based resource and service discovery for the web of things. In *2012 Sixth international conference on innovative mobile and internet services in ubiquitous computing*, pages 731–738. IEEE, 2012.

38 Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *FMCO*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010.

39 Eduard Kamburjan, Crystal Chang Din, and Tzu-Chun Chen. Session-based compositional analysis for actor-based languages using futures. In *ICFEM*, volume 10009 of *Lecture Notes in Computer Science*, pages 296–312, 2016.

40 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo: a session type toolchain for Java. *Science of Computer Programming*, 155:52–75, 2018. doi:10.1016/j.scico.2017.10.006.

41 Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003.

42 Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

43 Paul V Mockapetris. RFC 1035: Domain names - implementation and specification, 1987.

44 Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Composing services with JOLIE. In *ECOWS*, pages 13–22. IEEE Computer Society, 2007.

45 Dimitris Mostrous and Vasco T. Vasconcelos. Affine sessions. *Log. Methods Comput. Sci.*, 14(4), 2018.

46 Dimitris Mostrous and Vasco Thudichum Vasconcelos. Session typing for a featherweight Erlang. In *Proceedings of the 13th International Conference on Coordination Models and Languages (COORDINATION)*, volume 6721 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2011. doi:10.1007/978-3-642-21464-6_7.

47 Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *Proceedings of the 16th IFIP WG 6.1 Conference on Coordination Models and Languages (COORDINATION)*, volume 8459 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2014. doi:10.1007/978-3-662-43376-8_9.

48 Rumyana Neykova and Nobuko Yoshida. Let it recover: multiparty protocol-induced recovery. In *Proceedings of the 26th International Conference on Compiler Construction (CC)*, pages 98–108. ACM, 2017. URL: http://dl.acm.org/citation.cfm?id=3033031.

49 Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. *Logical Methods in Computer Science*, 13(1:17):1–30, 2017. doi:10.23638/LMCS-13(1:17)2017.

50 Barry Porter, Matthew Grieves, Roberto Rodrigues Filho, and David Leslie. REX: A development platform and online learning approach for runtime emergent software systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 333–348. USENIX Association, 2016. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/porter.

51 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science*, 13(2), 2017. doi:10.23638/LMCS-13(2:1)2017.

52 Jan S Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-OSGi: distributed applications through software modularization. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 1–20. Springer, 2007.

53 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In *Proceedings of the 31st European Conference on Object-Oriented Programming (ECOOP)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:31, 2017. doi:10.4230/LIPIcs.ECOOP.2017.24.

54 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019.

55 Filippo Visintainer, Leandro D'Orazio, Marco Darin, and Luciano Altomare. Cooperative systems in motorway environment: The example of Trento test site in Italy. In Jan Fischer-

Wolfarth and Gereon Meyer, editors, *Advanced Microsystems for Automotive Applications 2013*, pages 147–158, Heidelberg, 2013. Springer International Publishing.

56 Feng Xia, Laurence T. Yang, Lizhe Wang, and Alexey V. Vinel. Internet of things. *International Journal of Communication Systems*, 25(9):1101–1102, 2012. doi:10.1002/dac.2417.

57 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble protocol language. In *Proceedings of the 8th International Symposium on Trustworthy Global Computing (TGC)*, volume 8358, pages 22–41. Springer, 2014. doi:10.1007/978-3-319-05119-2_3.

Do Bugs Propagate? An Empirical Analysis of Temporal Correlations Among Software Bugs

Xiaodong Gu
School of Software, Shanghai Jiao Tong University, China

Yo-Sub Han
Department of Computer Science, Yonsei University, Seoul, South Korea

Sunghun Kim
Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong

Hongyu Zhang
The University of New Castle, Australia

Abstract

The occurrences of bugs are not isolated events, rather they may interact, affect each other, and trigger other latent bugs. Identifying and understanding bug correlations could help developers localize bug origins, predict potential bugs, and design better architectures of software artifacts to prevent bug affection. Many studies in the defect prediction and fault localization literature implied the dependence and interactions between multiple bugs, but few of them explicitly investigate the correlations of bugs across time steps and how bugs affect each other. In this paper, we perform social network analysis on the temporal correlations between bugs across time steps on software artifact ties, i.e., software graphs. Adopted from the correlation analysis methodology in social networks, we construct software graphs of three artifact ties such as function calls and type hierarchy and then perform longitudinal logistic regressions of time-lag bug correlations on these graphs. Our experiments on four open-source projects suggest that bugs can propagate as observed on certain artifact tie graphs. Based on our findings, we propose a hybrid artifact tie graph, a synthesis of a few well-known software graphs, that exhibits a higher degree of bug propagation. Our findings shed light on research for better bug prediction and localization models and help developers to perform maintenance actions to prevent consequential bugs.

2012 ACM Subject Classification Software and its engineering → Maintaining software

Keywords and phrases empirical software engineering, bug propagation, software graph, bug correlation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.11

Acknowledgements The authors would like to thank anonymous reviewers for their very insightful comments and constructive suggestions in greatly improving the quality of this paper.

1 Introduction

Software bugs are not isolated [30, 62], rather they may interact, affect each other, and trigger consequential bugs over certain software artifacts [38, 54, 56]. Identifying and understanding bug correlations could help developers localize bug origins, predict future bugs, and design better architectures to prevent bug propagation. Therefore, the study of bug propagation is of tremendous importance from both analysis and design points of view.

There has been much work on bug co-occurrence and localities, indirectly implying the propagation of bugs [24, 30, 48]. For example, Kim et al. [30] found that bugs do not occur in isolation, but rather in bursts of several related bugs. Zimmermann et al. [62, 63] investigate bug correlations over software dependency graphs and utilize the correlation effect for defect prediction. While these studies have shown the evidence of spatial correlation among bugs,

they do not investigate the existence of bug correlation across time steps. There are many unanswered questions about temporal correlations among the bugs. More specifically: 1) **Do software bugs propagate over time?** 2) **How do they propagate?** and 3) **Why do they propagate?**

In this paper, we investigate the temporal (a.k.a., time-lag) correlations among software bugs along with software evolution. We adopt the correlation analysis methodology from social science, which has shown effectiveness in identifying propagation of social phenomena (such as obesity and suicide) [12, 19]. Like in social network analysis [12], we define temporal correlation as the correlation between bugs across time steps.

In social science, corrections of social phenomena depends on social ties such as parents, relatives and friends. These social ties are usually represented as graphs (i.e., friend graph, neighbor graph, and sibling graph) [12, 50, 51]. Researchers then perform correlation analysis on these graphs [19, 12] to analyze the temporal correlation phenomenon. Choosing different graphs may lead to different findings. For example, obesity propagates more significantly on friend graphs than on sibling graphs [12]. Similar to social ties, there are also a various kinds of ties among software modules such as dependencies [41, 62, 63], collaborations [25, 45], and interactions [45]. We call such ties *software ties* which can be modeled by software graphs. For example, Zimmermann [62, 63] has built the dependency graph which characterizes data and call dependence between software artifacts and shows effectiveness in defect prediction. Pinzger et al. [45] proposed the developer-module network which models the relationships between developers and their contributions.

To explore the temporal correlation of software bugs on different software ties, we build graphs of three known artifact ties from the literature such as function calls [7], module inheritance [22], and file co-change [14, 24]. Nodes in each graph stand for source code files, while edges refer to software ties such as calling, inheritance, and co-change (CC) relationships. We build software tie graphs for four open-source projects (HTTPClient, Jackrabbit, Lucene, and Rhino) and then perform the longitudinal logistic regression [12] on the bug and metric statistics on these graphs. We measure the temporal correlation among bugs using the coefficients of the longitudinal logistic regression (LRC) and compare the results with well-recognized propagation phenomena such as obesity [12] and happiness [19].

Our experimental results show that there are significant temporal correlations among bugs through the three basic software ties. The logistic regression coefficients (LRC) of the three basic graphs are 0.19, 0.32 and 0.39, respectively, showing far more compelling results than the random graph (LRC=0) and are generally more significant than social events such as happiness (LRC=0.12) and obesity (LRC=0.28) [12, 19]. This answers our questions raised in the beginning: **bugs can propagate along with software evolution, through common software ties, and the propagation can be significant.** Having had more buggy neighbors among these ties in the past increases the likelihood of being buggy in the near future.

Based on the experimental results, we propose a hybrid graph (tie) which synthesizes basic graphs by merging and selecting their edges according to their temporal correlation. The proposed hybrid graph exhibits a higher degree of temporal correlation than basic graphs, with an average LRC of 0.66, which is greater than the values of basic graphs. That means, the hybrid graph provides a more efficient way for identifying temporal correlations among bugs.

The properties of temporal bug correlation have many benefits for developers. For example, using the correlation property, developers can link related bug reports, predict potential consequential bugs, and perform maintenance actions to prevent bug propagation.

Developers can also improve the quality of software by designing better artifact structures which have less bug correlation. The software ties between temporal correlated bugs can facilitate the localizing of correlated bugs and the understanding of their root causes.

Overall, our study makes the following contributions:

- We evaluate the temporal correlation of software bugs on three software ties (graphs), and show that software bugs are indeed temporally correlated over the ties. To our best knowledge, it is the first work that investigates the phenomenon of temporal correlation among software bugs.
- We design a hybrid graph and show that the graph highlights the bug correlation effects more significantly.
- We discuss reasons to the temporal bug correlations and suggest possible applications of our findings.

The rest of this paper is organized as follows. Section 2 discusses the background. Section 4 presents the common setup for bug correlation analysis. Section 5 presents the results for basic graphs, hybrid graph as well as the reasons of temporal bug correlation, followed by discussions in Section 6. Section 7 presents the related work, and Section 8 concludes the paper.

2 Background

This section introduces the background of *temporal correlation analysis* including the concepts, measures, and applications.

2.1 Temporal Correlation Analysis in Social Science

The temporal correlation analysis aims at identifying the consequential occurrence of a phenomenon in social science [12, 4]. For example, Christakis and Fowler [12] found that obesity can be "contagious" among various social ties such as friends and siblings. They evaluated the spread of the body-mass index in a densely interconnected social network of 12,067 people throughout a 32 year period and found weight gain in one person was associated with weight gain in his or her friends, siblings, spouses, and neighbors. Being overweight is also identified to be correlated among friends in schools [53]. Researchers estimated peer effects for adolescent weight using data from the *National Longitudinal Study of Adolescent Health*, and found that mean peer weight is correlated with adolescent weight especially among females.

Correlation analysis has also shown to be effective in studying online social networks such as Facebook, MySpace and Flickr [51]. For example, Anagnostopoulos et al. [4] identified social influence by investigating correlations between individuals in an online social network, and proposed measures to assess the correlations. Chenhao [50] measured the individual influence among online friends. They designed a social action tracking algorithm to assess the quantitative values of influence and utilized them to help predict users' future actions. Kempe [29] considered temporal correlation as a way to choose the most influential sets that can maximize the spread of information.

2.2 Modeling Ties Using Graphs

Graphs, as desirable models to capture ties, are widely used for temporal correlation analysis [12, 50, 51]. We list several graphs used in different research areas in Table 1. In online social networks such as Flickr, users are denoted as nodes while ties such as friendships and

Table 1 Graphs models for correlation studies in other areas.

Area	Graph	Node	Edge	Weight
social science [12]	obesity	individual	friend	–
social science [19]	happiness	individual	friend	–
data mining [4]	Flickr	user	friend	influence

corporations are represented as edges [4]. In the social science literature, subject persons are denoted as nodes (egos and alters), while social ties such as friends, spouses, and siblings are represented as edges.

Identifying appropriate ties is critical in temporal correlation analysis. Different social ties correspond to different graphs and show different degrees of correlation, affecting the results of correlation analysis. For example, obesity exists in friend graphs but not in neighbor graphs [12]. Happiness is temporally correlated among neighbors but not among co-workers [19].

2.3 Measuring Propagation

Most propagation studies identify the propagation of a behavior by measuring temporal correlations between neighbors and egos in the tie graph. Intuitively, if a behavior propagates from one node to another, then the neighbor (the affected node) can also influence the ego (the source node). In that sense, the behavior occurrence between the neighbor and the ego can have a high correlation [4, 6]. In other words, propagation exists only if there exist correlations among egos and neighbors.

Such temporal correlation can be identified using the longitudinal logistic regression (LLR) [9, 12]. The LLR is a logistic regression model for longitudinal data (i.e., data that tracks the same subjects at multiple points in time). It assumes that the ego's behavior status at any given time t is a function of various attributes, including intrinsic features of the ego (e.g., age, sex and education level), any previous behavior of the ego at time t$-$1 and neighbors' disease status at times t$-$1 and t. Let n^{t-1} denote the number of neighbors that are affected at time t$-$1, y^t denote the ego's affection status (1=affected, 0=otherwise) at current time t, and x_i^t (i=1...N) denote the ego's intrinsic metrics at time t. The probability that the ego is affected at present $p(y^t)$ can be estimated as:

$$ln(\frac{p(y^t)}{1-p(y^t)}) = \boldsymbol{\alpha} n^{t-1} + \beta_0 + \beta_1 x_1^t + ... + \beta_N x_N^t \quad (1)$$

where y^t stands for the dependent variable; n^{t-1} and all x^t's are independent variables. α and β represent the regression coefficients for each independent variable [4].

The longitudinal regression model is commonly solved by the generalized estimating equation (GEE) [34, 46] which accounts for multiple observations of the same ego across examinations and across ego-neighbor pairs. By running GEE on the LLR model, we obtain the coefficients (e.g., α, β) for Equation 1. The temporal correlation is usually measured by the first coefficient α which indicates the correlation between y^t and n^{t-1} in Equation 1. We denote α as LRC (logistic regression coefficient) [9, 12]. The higher the LRC, the more significant correlation between the two variables. LRC is a widely used measure for temporal correlations between social behaviors [19]. For example, happiness is shown to propagate among social ties with an LRC of 0.12 [19], while obesity among social ties shows an LRC of 0.28 [12]. In a random network where behaviors are independently generated, the average LRC is expected to be 0.

Table 2 Overview of software tie graphs.

Graph	Node	Edge	Weight
function call graph	source files	function call	# call sites
type hierarchy graph	source files	inheritance	–
co-change graph	source files	co-change	# co-changes

It is worth noting that in some areas, having correlations between behaviors does not necessarily indicate the existence of propagation [12]. The correlations could come from homophily or confounding factors (i.e., two nodes share the same characteristics) [4, 6, 12]. For example, in the obesity studies, people may tend to make friends with others who have the similar weights. This suggests the need to distinguish the homophily and confounding factors from correlations according to different scenarios. One commonly used method to address this issue is the edge-reversal test [4, 12]. This test first reverses the direction of all edges and then run the longitudinal regression on the reversed graph. Since the homophily and confounding factors are only based on the fact that two nodes often share the same characteristics. These factors must be independent of the two individuals' identification of the other as a neighbor. Therefore reversing the edges will not change the behavior correlation estimate significantly [4]. In other words, if correlations change significantly by reversing the graph, we can exclude the homophily and confounding factors.

3 Study Methodologies

This section presents our study methodologies for identifying the temporal correlations among bugs. We first introduce the software tie graphs where temporal correlations are dependent on, followed by the measurement of temporal correlations.

3.1 Software Tie Graphs

Similar to common social ties such as friends, siblings, and spouses (Section 2.2), software artifacts also have a various kinds of ties such as dependencies [41, 62, 63], collaborations [25, 45], and interactions [45]. Graphs, as effective models of social ties, are also widely used to represent software artifacts' ties. For example, Zimmermann et al. [62, 63] represented dependencies between software binaries by introducing the dependency graph. They further observed a substantial correlation between central binaries and defects on the graph. Pinzger et al. [45] leveraged graphs to model developer-modules and found correlations between centrality measures of developer-module networks and failure-prone modules.

We define a *tie graph* as a directed graph $G = (V, E, w)$ where V denotes the set of nodes (source files), $E \subseteq V \times V$ is the set of edges (artifact ties), and $w : E \to \mathbb{R}$ represent the weighting function for each edge. We select three software artifact ties according to well-known defect factors in the software engineering literature [15, 21, 32, 36, 45, 62, 63], and build three corresponding graphs: *function call graph*, *type hierarchy graph*, and *co-change graph*. An overview of all software artifact tie graphs is presented in Table 2.

3.1.1 Function Call Graph

The function call graph models the caller-callee relationship between artifacts. Intuitively, bugs from the callee could affect the caller since the caller reuse its codes directly. If a function in file A calls another function in file B, A and B are represented as two nodes in

the graph, and a directed edge from node A to node B is added. The weight of each edge is the number of call sites between nodes.

Figure 1 (a) and (c) illustrate an example of call graph. The class *Signature* has a function *sampleMethod* which calls a function in object of another class *JavaTextLabelProvider*. Then, there are two nodes *Signature.java* and *JavaTextLabelProvider.java* in the call graph and an edge from node *Signature.java* to the node *JavaTextLabelProvider.java*. The weights (numbers in the edges) indicate the total numbers of call sites from a file to another file.

3.1.2 Type Hierarchy Graph

The type hierarchy graph captures the inherit relationships among entities. Type hierarchy indicates the inheritances and implementations among classes. It is not only an important representation of software design, but also reveals substantial dependencies among artifacts. Nodes in type hierarchy graphs also stand for files. If a class in file A extends a class in file B, a directed edge from A to B is added. The weight of each edge is 1 since there is no multiple inheritances between nodes. Figure 1 (a) and (d) show an example of type hierarchy graph. The class *Signature* extends another class *AbstractSignature*. Then, there are two nodes *Signature.java* and *AbstractSignature.java* in the hierarchy graph and an edge from node *Signature.java* to the node *AbstractSignature.java*.

3.1.3 Co-Change Graph

We also use co-change graphs to capture the co-change relationships among files. Files that were changed together have implicit couplings [14, 24]. Co-changed files could have relevant functionality, or be maintained by the same developer. In a co-change graph, nodes stand for source files. If two files were changed at the same time, an edge weighted with co-changed times will be connected to both files.

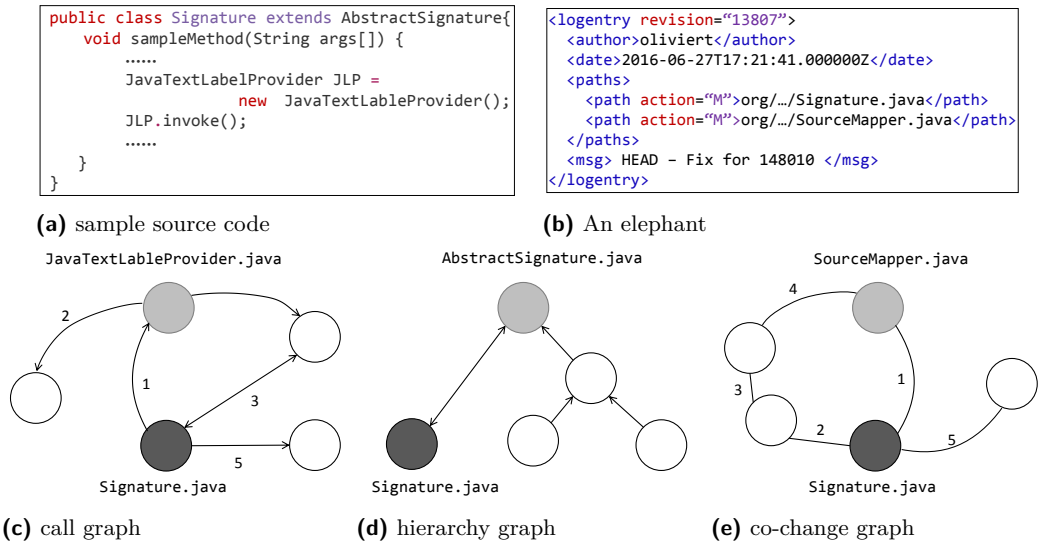
Figure 1 (b) and (e) illustrate an example of co-change graph. The file *Signature.java* was changed together with the file *SourceMapper.java*, therefore, there is an undirected edge from node *Signature.java* to node *SourceMapper.java*. The weights (numbers in the edges) indicate the total numbers of changes that two files were changed together.

3.2 Measuring Temporal Bug Correlations

We follow the longitudinal logistic regression methodology introduced in Section 3, and measure bug propagation by estimating the time-lag bug correlations between neighbors and egos across each consecutive pair of examination points (i.e., time t-1 and t). Specifically, we perform longitudinal logistic regressions (LLR) (Section 3) on the bug occurrence data and measure the time-lag bug correlations using the regression coefficient (LRC) (Section 2.3). Let $y^t = \{y_l^t\}_{l=1}^N$ denote the status of bug for N data instances at examination point t where $y_l^t \in \{1, 0\}$ (1=buggy, 0=clean). Let $x^t = \{x_l^t\}_{l=1}^N$ denote the corresponding independent variables, where $x_l^t = \{x_{1l}^t, x_{2l}^t, ..., x_{Ml}^t\} \in R^M$ is a collection of common metrics that affect the presence of bugs such as lines of codes (LOC) and cyclomatic complexity (CYC) for instance l at examination point t. Let $n^t = \{n_l^t\}_{l=1}^N$ denote the number of buggy neighbors for each instance l at examination point t. The LLR model fits a logistic regression form [4]:

$$p(y_l^t = 1) = \frac{e^{\alpha n_l^{t-1} + \beta_0 + \beta_1 x_{1l}^t + ... + \beta_M x_{Ml}^t}}{1 + e^{\alpha n_l^{t-1} + \beta_0 + \beta_1 x_{1l}^t + ... + \beta_M x_{Ml}^t}} \quad (2)$$

where y_l^t represents the dependent variable, n_l^{t-1} and all x_l^t's denote independent variables, and M stands for the number of independent variables for controlling. All independent



(a) sample source code **(b)** An elephant

(c) call graph **(d)** hierarchy graph **(e)** co-change graph

Figure 1 Examples of source files and artifact tie graphs.

variables we select are shown in Table 3. We collect source code metrics (i.e., LOC, CYS, and ESS) using the Understand tool [1] and changing metrics (i.e., NOA and NOC) from the SVN commit logs.

4 Experimental Setup

This section describes the experimental setup. We first present our research questions. Then, we introduce the data sets, implementations, as well as baseline graphs.

4.1 Research Questions

We design our experiments to address the following research questions:

- **RQ1: (Temporal bug correlations on known software ties) Do software bugs propagate through the known software ties?** We build graph models for known software artifact ties such as call, inheritance, and co-changing. They are commonly used graphs to capture artifact dependencies [41, 62, 63] and collaborations [25, 45]. We perform longitudinal logistic regressions to measure the time-lag correlations (Section 2.3) between artifact ties. Then, we compare the results (LRC) with well-recognized conrrelation results in other research areas.
- **RQ2: (Temporal correlations on proposed graph) Do software bugs propagate across the proposed hybrid graph?** Based on the observation of basic graphs, we propose a hybrid graph which synthesizes basic graphs according to their correlation properties. We measure bug propagation across the proposed hybrid graph.
- **RQ3:(Casual mechanisms) What are the reasons of the temporal bug correlation phenomenon?** To explain the causal mechanism of such temporal correlations, we conduct a qualitative analysis in real software development and show a few reasons.

Table 3 A taxonomy of the independent variables.

	metric	description	rationale
controls	LOC [32]	Lines of code	Large components are more likely to be defect-prone [32].
	CYC [36]	Sum of cyclomatic complexity of all nested functions or methods	More complex components are likely more defect-prone [36].
	ESS [36]	Sum of essential complexity for all nested functions or methods	
	NOA [21]	Number of authors	Components with many unique authors likely lack strong ownership, which in turn may lead to more defects [21].
	NOC [21]	Number of changes	The number of changes to code in the past was a successful predictor of faults [21].
	y_l^{t-1}	Number of defects in prior examination for instance l.	Defects may linger in components that were recently defective [21].
test	n_l^{t-1}	Number of neighbor bugs previously for instance l	Our hypothesis

Table 4 Summary of subject projects.

dataset	HTTPClient	JCR	Lucene	Rhino
time span	2007.4–2011.12	2006.12–2011.12	2010.3–2011.12	1999.4–2012.6
# unique files	281	1134	511	388
# total bugs	576	938	696	302
exam. points (selected releases)	4.0, 4.0.1, 4.1α2, 4.0.2, 4.1.1, 4.1.2, 4.2α1	1.1.1, 1.2.1, 1.3, 1.3.3, 1.4, 1.4.3, 1.4.6, 1.5, 1.5.5, 1.6, 2.0, 2.1, 2.2, 2.2.5, 2.2.7, 2.3, 2.3.3	3.0.1, 3.0.2, 3.0.3, 3.1, 3.2, 3.3, 3.4, 3.5	1_4r3, 1_5r1, 1_5r2, 1_5r4, 1_5r5, 1_6r1, 1_6r6, 1_7r2, 1_7r3, 1_7r4

4.2 Subjects and Data Collection

We collect bug repositories based on Herzig et al.'s manually verified bug database [23]. We select four projects[1] from their data sets: HTTPClient, Jackrabbit (JCR), Lucene, and Rhino. They are widely used in the software engineering literature, and more importantly the data sets have been manually verified [27]. Table 4 shows an overview of these data sets. They have various project periods (from 2 to 13 years) and different scales and bug densities. The HTTPClient contains only 281 distinct files with 576 bugs in total while Jackrabbit contains 1134 files with less than 1000 bugs.

Inspired by the study of obesity propagation [12], we separate the whole evolution of a project (i.e., all revisions) into different time periods by selecting several revisions as examination points ($t = 1, 2, ..., T$, where T is the total number of examination points) (see Figure 2).

[1] For one of the five bug datasets, i.e., Tomcat, the source code of corresponding releases is not available.

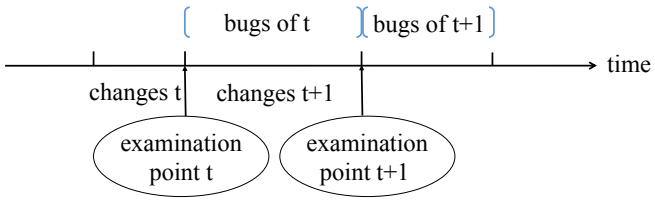


Figure 2 Illustration of examination points.

To make the examination points more reasonable, we select revisions when a release was published and includes bug fixes according to the release history. Different releases may have different purposes. For example, HTTPClient 4.2 is just released for enhancement while HTTPClient 4.1.3 is for bug fix. We cannot identify bugs during periods that contains no bug fix assignments. Therefore, in this example, we select as examination point the revision when release 4.1.3 was published. In order to balance the intervals between examination points, we also remove examination points that are too close to another (i.e., less than one month). We list all the selected examination points in Table 4.

At each examination point t, we collect JAR files of the corresponding release and capture a snapshot call graph and a hierarchy graph based on the JAR files. We generate all call and hierarchy graphs using the WALA tool (1.4.3) [2] with the default options.

We collect co-change information by parsing the commit history. For each examination revision t, we collect co-changed files in the previous examination periods (t-1 to t). If two files were committed at the same time during that period, we connect them with an edge in the corresponding co-change graphs.

Then, we label files at an examination point t by collecting bugs fixed at interval (t to t+1). For each bug in a data set, we compare the bug id against the messages in SVN commit logs to get the link between the bug and its fix revision. Then, we link the bug to files committed at that revision. For each file at each examination point (at time t) we count the numbers of fixed bugs during the examination period (t to t+1). If the number is more than zero, we label the file as buggy.

4.3 Implementation Details

We extract artifact metrics such as LOC and CYC using the Understand code analysis toolkit [1]. We implemented the LLR model using Matlab. The LLR Model is then estimated by solving a general estimating equation (GEE) with an equicorrelated working covariance structure [34]. We perform GEE using the GEEBOX tool [46] in Matlab.

4.4 Comparison

We compare the temporal correlation of software bugs in software ties with widely-accepted results in other research areas. We first compare the correlation value (LRC) to a randomly constructed graph where bugs are evenly distributed to the nodes. By mathematical definition [10], the LRC in a random graph is 0.0. LRC= c ($c > 0$) means that the log odds of being buggy at time t increase by c times of the number of buggy neighbors increases at time t-1. For more references, we also compare bug correlation with correlation results from the social science literature such as obesity and happiness. Obesity is temporally correlated with the numbers of opposite sex siblings who are obese (LRC=0.28) [12] and happiness is temporally correlated with the numbers of neighbors who are happy (LRC=0.12) [19].

Table 5 Results of time-lag bug correlation ('.' stands for results with p-values≥0.05).

graphs	dataset	independent variables						
		n^{t-1} (**LRC**)	y^{t-1}	loc	cyc	ess	noa	noc
function call	HTTP	**0.38**	.	.	.	.	.	.
	JCR	**0.13**	0.01	0.01	-0.01	0.01	-0.07	0.20
	Lucene	**0.10**	.	0.00	.	0.01	0.26	.
	Rhino	**0.16**	.	0.01	-0.02	.	.	0.08
	Avg	**0.19**	0.01	0.01	-0.02	0.01	0.09	0.14
type hierarchy	HTTP	.	.	.	.	.	.	.
	JCR	**0.23**	0.004	0.01	-0.01	.	-0.05	0.17
	Lucene	**0.41**	.	.	.	0.01	0.27	0.06
	Rhino	.	.	0.004	-0.01	.	.	0.07
	Avg	**0.32**	0.004	0.01	-0.01	0.01	0.11	0.1
co-change	HTTP	**0.51**	.	.	.	.	.	.
	JCR	.	.	0.004	.	.	.	0.19
	Lucene	**0.30**	.	.	.	0.01	0.44	-0.19
	Rhino	**0.36**	-1.56	.	.	.	.	.
	Avg	**0.39**	-1.56	0.004	.	0.01	0.44	0

5 Results

This section presents our experimental results addressing the research questions (Section 4.1).

5.1 Bug Correlation on Known Software Ties (RQ1)

Table 5 shows the results of the logistic regression coefficients on known software artifact ties. Values in each row are the coefficients for different independent variables. We note '.' for the values without statistical significance (p≥0.05), since they are meaningless [13]. The first column of these coefficients is the LRCs. The bold values indicate LRCs that are higher than that of the baselines. For example, the LRC for Rhino in the co-change graph is 0.36. That means, having one more buggy neighbor at time t-1, the ego's risk (i.e., log odds) of being buggy at time t is 0.36 much higher.

As the results indicate, the call, hierarchy and co-change graphs have high positive LRCs in most subjects. The average LRCs for them are 0.19, 0.32, and 0.39, respectively. These values are much greater than the random graph (0.0) and are comparable to typical social correlations (obesity 0.28 and happiness 0.12). That means that temporal bug correlation is present on these graphs.

In summary, our results show that on some known ties (calling, extending and co-changing), software bugs show evidence of temporal correlation. There are correlations between a bug in one time period and bugs in the neighbors in the next time period.

Software bugs are temporally correlated in some known ties such as call, type hierarchy and co-changes.

5.2 Bug Correlation on Hybrid Tie (RQ2)

As the results in Section 5.1 indicate, different graphs may have different degree of bug correlation. We were wondering it is possible to synthesize a hybrid graph (software tie) that exhibits higher degree of correlation. Then, with the proposed hybrid graph, developers could identify subsequent bugs more efficiently. We propose to build a hybrid graph based

Table 6 Results for the edge-reversal test ('-' stands for results with p-values≥0.05; '*' means that the result shows a significant change after reversing edges.).

graphs	dataset	independent variables						
		n^{t-1} (**LRC**)	y^{t-1}	loc	noc	noa	ess	cyc
function call	HTTP	0.38	-	-	-	-	-	-
	JCR	**0.06**(*)	0.01	0.01	0.21	-0.06	0.01	-0.01
	Lucene	0.16	0.02	-	-	0.26	0.01	-
	Rhino	-(*)	-	0.01	0.08	-	-	-0.02
type hierarchy	HTTP	-	-	-	-	-	-	-
	JCR	-(*)	0.004	0.01	0.19	-	0.01	-0.01
	Lucene	-(*)	-	-	0.08	0.27	0.01	-
	Rhino	-	-	0.004	0.07	-	-	-0.01
hybrid	HTTP	**0.61**(*)	-	-	-	-	-	-
	JCR	-(*)	0.004	0.01	0.19	-	0.01	-0.01
	Lucene	-(*)	-	-	-	0.29	0.01	-
	Rhino	0.44	-0.5	0.01	0.08	-	-	-

on bug correlation on basic graphs and evaluate temporal bug correlations on the proposed graph using approaches in Section 3.2.

5.2.1 Synthesizing Hybrid Correlation Graphs

Similar to the process of machine learning, we split all releases of a project into a training and a test set. We estimate LRCs for basic graphs in the training set. Then, in the test set, we synthesize a hybrid graph for each release based on the new basic graphs and their historical LRCs (i.e., LRCs in the training set).

We use a "greedy expansion" strategy when synthesizing the hybrid graph. We initially select the "best" graph (with the highest LRC) from basic graphs. Then, we "optimize" edges in the selected graph as follows: for edges in the "best" graph, if an edge appears in more than one basic graphs, we keep it in the hybrid graph. For edges in other basic graphs, if an edge appears in more than one basic graph, and the total LRC of other basic graphs is greater than that of the selected "best" graph, we add this edge in the hybrid graph as well.

Algorithm 1 illustrates the pipeline of constructing the hybrid graph. In the training stage, we compute LRCs for basic graphs as described in Section 3. In the test phase, we initially assign weights to edges in the new basic graphs using their historical LRCs (i.e., LRCs in the training set). Then, we merge their edges and add up the corresponding edge weights. Finally, we remove edges whose weights are no more than the maximum LRC in the merged graph. The remaining merged graph is then returned as the hybrid graph.

Figure 3 shows an example of the synthesis of hybrid graph. The top three graphs are basic graphs constructed from a project with historical LRCs of 0.6, 0.1, and 0.3, respectively. We also assign edge weights as 0.6, 0.1, and 0.3. To synthesize the hybrid graph, we first merge the edges together and add up the corresponding edge weights. Then, we select edges from the merged graph (bottom left) whose weights are greater than 0.6 (i.e., 0.7 and 0.9). Finally, these selected edges as well as the original nodes constitute the hybrid graph (bottom right).

[2] The c, h, and cc are short for call graph, hierarchy graph, and co-change graph, respectively

Algorithm 1 Constructing hybrid bug correlation graph.

Input:
Basic Graphs, $G^i = (E^i, V, w^i), (i \in \{c, h, cc\})^2$
Historical LRCs for basic graphs, $LRC(G^i)$
Output: Hybrid Graph, $G^* = (E^*, V, w^*)$

Choose the maximum LRC from all G^i

$$LRC_{max} = \max_{G^i \in \{G^c, G^h, G^{cc}\}} LRC(G^i); \tag{3}$$

$E^* = E^c \cup E^h \cup E^{cc}$
for all $(u, v) \in E^*$ **do**
  $w^*(u, v) = w^c(u, v) + w^h(u, v) + w^{cc}(u, v)$
  if $w^*(u, v) \leq LRC_{max}$ **then**
    $E^* = E^* \setminus (u, v)$
  end if
end for
$G^* = (E^*, V, w^*)$;
return G^*

5.2.2 LRC on The Hybrid Graph

After constructing the hybrid graph for each release, we estimate temporal bug correlations on the proposed graphs using the same methodology (Section 3.2).

Table 7 shows LRCs for hybrid graphs. For each project, we set the first third of releases for training and the remaining releases for test. Each row shows LRCs of a specific graph for different datasets. The last column shows the average LRCs (excluding LRCs with $p > 0.05$) for each graph.

As shown in the results, the hybrid graph exhibits much higher LRCs in most subjects (1.09 in HTTPClient, 0.65 in Lucene, and 0.52 in Rhino). The average LRC for the hybrid graph is 0.66, much higher than those of call, hierarchy, and co-change graphs (0.18, 0.38 and 0.35, respectively). Table 6 shows the results for edge-reversal test. We reverse all the edges and present the coefficients again. Each line represents coefficients in terms of different independent variables. The first column shows the new LRCs for hybrid graph. The results show that after reversing all the edges, the LRCs change significantly in three out of four subjects.

The hybrid graph in Jackrabbit shows a maximized yet not improved LRC. We find it is constituted by the call graph and the hierarchy graph since the co-change graph in this subject shows no significant bug correlation. Then, since hierarchy graphs are relatively sparse and may have fewer intersections with call graphs. The resulting hybrid graph can be very sparse. On the other hand, both the call and the hierarchy graphs represent source code based dependencies. Their small intersections may have no information gain without combining other category of graphs such as co-change graphs. These could be the main reasons for such an outlier.

In summary, the degree of temporal bug correlation is much higher in the proposed hybrid graph than that in other basic software ties. The results indicate that the proposed hybrid graph can exhibit bug propagation more effectively.

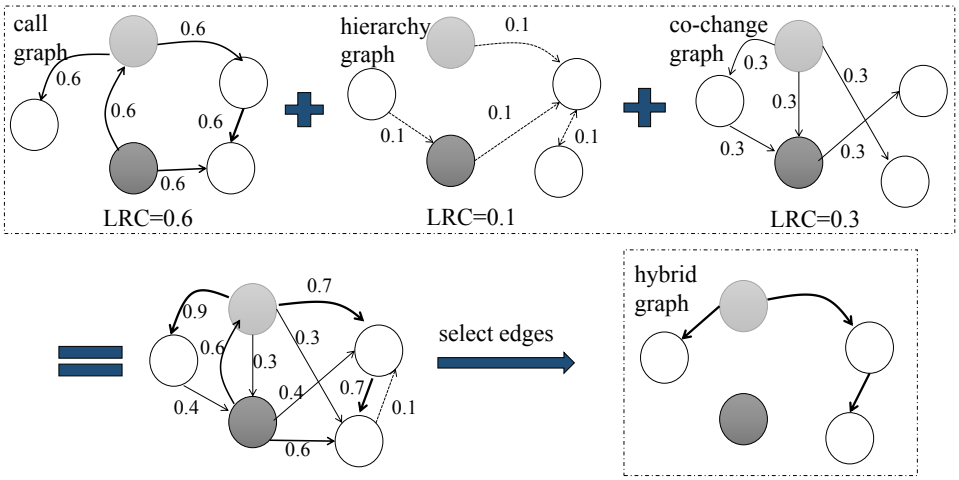


Figure 3 Hybrid graph construction.

Table 7 Results of temporal bug correlation in the proposed hybrid graph. The bold numbers mean higher degree of correlation than basic graphs.

Graph	HttpClient	JCR	Lucene	Rhino	**Avg**
call	.	0.15	0.13	0.27	0.18
hierarchy	.	0.36	0.39	.	0.38
co-change	0.45	.	0.30	0.29	0.35
hybrid	**1.09**	0.36	**0.65**	**0.52**	**0.66**

The proposed hybrid graph provides a synthesized software tie that exhibits a higher degree of temporal bug correlation.

5.3 Casual Mechanisms behind the Temporal Bug Correlation (RQ3)

In this section, we explain the casual mechanisms behind temporal bug correlation through case studies.

The most fundamental and challenging question here is why bugs propagate. Bugs seem to be statically created at different times. How do they correlated between different artifacts?

To explain the causal mechanism of such time-lag correlations, we conduct a qualitative analysis in real software development. Specifically, we select some real bugs from our data set and analyze how bugs of one code can "propagate" to another piece of code. According to the bug reports and issue discussions, we identified three mechanisms for bug correlations.

1) Code Reuse: A possible causal mechanism for the temporal correlation could be the code reuse [47] (e.g., clone, inheritance, components, template, framework) on dependent artifacts. Dependent artifacts such as method calling and inheritance have high couplings [17]. They may reuse the codes or logics directly. Therefore, bugs can propagate when an artifact invokes or extends another piece of code directly. For example, the bug LUCENE-3026 was caused by declaring a variable as short in *SegGraph.java*. This bug spread to its offspring *BiSegGraph.java* which reuses the same code, resulting in bug LUCENE-3049. If the correlated bug could be detected earlier, there would be no need to report the second bug again.

2) Bugfix: Bugfix can be another way to propagate bugs on dependent artifacts [43]. A bugfix for one piece of code may affect a dependent piece of code which has the same function and appears to invoke or co-change frequently with the ego [40]. For example, LUCENE-3505 was caused by *BooleanScorer2.freq()*. But the fix affected a dependent file *DisjunctionSumScorer.java*, leading to LUCENE-4401. In the discussion of LUCENE-4401, the developer mentioned such causality:

> *"I committed (also backported to 3.x branch, the bug does not affect any releases but would have affected unreleased 3.6.2 code, as it was caused by my previous bugfix: LUCENE-3505)."*

The same causality happened in LUCENE-3631 and LUCENE-3855. As the developer mentioned in their discussion of LUCENE-3855:

> *"I (accidentally!!) caused this with LUCENE-3631, where we moved writable deletes from SegmentReader into IndexWriter."*

3) Human Factors: Another explanation of temporal bug correlations could be the developer interactions among artifacts [55, 18]. Bugs might propagate because the same careless developer makes the same mistake or clones buggy codes to elsewhere [48, 24]. For example, both LUCENE-2478 and LUCENE-3446 are bugs of missing *NullPointerException* checking. The LUCENE-2478 was found on May, 2010. The developer forgot to check null return of *Filter.getDocIdSet()* in the *CachingWrapperFilter.java*. Later, a similar mistake (LUCENE-3446) by the same developer was found on Sept, 2011. He did not check null return in the *BooleanFilter.java* again. We found both files had been changed together in revision 722174 in 2008. If we could predict the correlated bug and carefully test it in the *BooleanFilter.java*, the second bug would not happen.

A more sophisticated empirical study on the percentage of each mechanism could be added in future work.

The temporal correlation among software bugs can be caused by code reuse, bugfix, and human factors.

6 Discussion

6.1 Application of Findings

In this section, we discuss potential applications of temporal bug correlations for practitioners. Our study and findings can inspire several research directions in the future.

- Defect Prediction: Traditional defect prediction techniques usually train a machine learning classifier with artifact metrics and make predictions by classifying new buggy instances. As Figure 4 summarizes, traditional defect factors include artifact metrics such as source code metrics (i.e.,size, complexity), churn, authors [15, 32, 36]. These factors are related to individual artifacts and are time independent. Besides these traditional factors, researchers also found that bugs can be time-dependent. In other words, defects may linger in components that were recently defective [21]. We refer to the latter category of factor as "dependence". In this paper, we have identified the third factor – temporal correlation. This means that neighbor bugs at previous time step may influence ego's bug status. The temporal bug correlation and hybrid graph suggest a new factor to

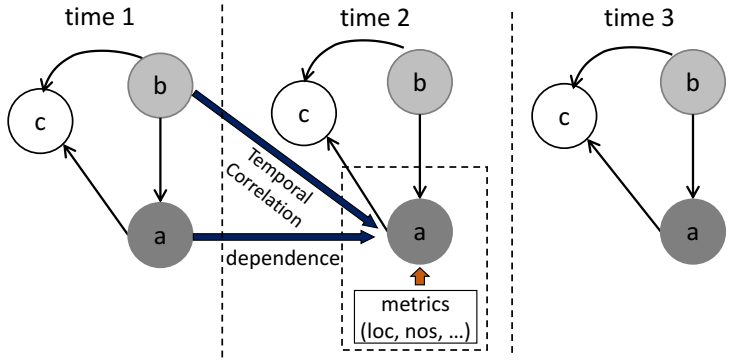


Figure 4 An illustration of factors to software defects. The three graphs from time 1 to time 3 represent three hybrid graphs in consecutive stages. Each circle stands for an artifact. The bold arrows represent different defect factors.

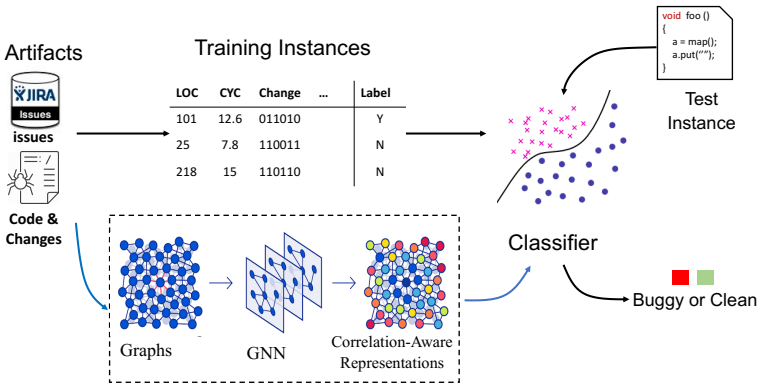


Figure 5 An example of the propagation based defect prediction.

software defect prediction. We can enhance existing defect prediction model by combining temporal correlation with traditional defect metrics (i.e., LOC, Complexity). Besides simply adding temporal correlation as a new metric, we can apply the graph neural network (GNN) [31] to the proposed hybrid graph. GNN is a popular deep learning model for learning graph representations. As our hybrid graph incorporates significant information about bug correlations, it can be utilized to help the learning of defect prediction models. For example, a defect prediction model (Figure 5) could be developed, which runs the graph neural network on the hybrid graph to learn a correlation-aware representation for each artifact. The learnt representation, together with traditional metrics, are taken as input to the classifier for the prediction.

- Linking Correlated Bug Reports: Another possible application is to link correlated bug reports. Identifying related bug reports have been a key issue in software maintenance [20, 49, 42, 42]. Researchers have developed a variety of approaches to retrieve semantically related bug reports [49, 42]. With the property of temporal bug correlation, developers can link several inter-related bug reports based on both temporal and semantic correlation of bugs for efficient and effective fault diagnosis. For example, a series of bugs could be reported due to temporal correlations, where fixing one can help resolve others. Moreover, the links may represent other types of relevance such as re-occurrence or co-occurrence

between two bugs. Recently, researchers have developed methods to predict semantically linkable Stack Overflow posts [57] and identify linked incident reports for online service systems [11], which have all confirmed the usefulness of linked knowledge.

- Bug Localization. Bug localization is one of the core tasks for software maintenance. Existing bug localization approaches often consider individual bug reports and identify buggy modules based on text similarities [33]. Some work also consider the historical similar bug reports [60], co-change histories [52], and the number of previous bugs [16]. The results of this research can provide more information, especially the temporal correlation information, to the existing bug localization work. Such information can help track the causal paths of bug and pinpoint their root cause, thus facilitating bug localization and triage.
- Architecture Refactoring. The proposed hybrid graph provides a new perspective on software design and quality management. Developers can avoid temporal bug correlation in architectural design in order to improve the quality and reliability of software. For example, they can estimate the temporal correlations and then construct the hybrid correlation graph using our LLR-based analysis tool. Then, they can adjust design paradigms (e.g., UML graphs) by removing unnecessary artifact connections that have edges in the hybrid graph.

6.2 Threats to Validity

As a proof of concept, all projects investigated in our experiments are developed as JAVA open source projects. Although Java is one of the most popular programming languages, it might not be representative of commercial projects and projects written in other languages. However, our study is not limited to a certain language as it is conducted on software graphs which can be extracted from most languages. Investigating temporal bug correlations in other languages remains our future work.

Another threat lies in the selection of examination points. We selected examination points according to the selected releases. In fact, in some data sets, the intervals between releases vary a lot, leading to graphs that are either too dense or too sparse. We mitigated the threat by removing releases that are too close to others. Yet the intervals are still not equal. The selection of releases might affect our experimental results, however although the examination points are not selected with equal intervals, our results are still valid. This is because examination intervals for different subjects vary a lot but most of the results show the same trends. This means that altering the examination points does not change the results significantly.

7 Related Work

7.1 Bug Distribution and Dependence

Besides our work, there have been other studies that investigate bug distributions and find similar phenomena of this sort. For instance, Zimmerman et al. [61] analyzed the relationships between failures and dependent neighborhoods on dependence graphs. They found that depending on a component that has failures does not have an effect on failures of the dependent component in VISTA. Andersson [5] investigated how faults in large software systems are distributed over modules and discovered that the distribution of faults over modules follows the Pareto principle [28]. Zhang [59] found a Weibull distribution of bugs across packages in Eclipse system. Murgia et al. [40] represented Java software as artifact tie graphs and analyze the relationship between graph properties, statistic of software metrics,

and the distribution of bugs in such graphs. They found that the distribution of bugs across compliant units exhibits a power-law behavior in their tails, suggesting the spread of bugs in the software system. Mondal et al. [39] investigated bug propagation through code cloning. They define clone evolution patterns that reasonably indicate bug propagation through code cloning and found that up to 33% of the clone fragments that experience bug-fix changes can contain propagated bugs. Ai et al. [3] proposes a new software network model and analyzes the relationship between nodes or defects distribution and software network parameters, as well as high-risk module excavation through a defect density analysis.

Different from these studies, we conduct a deeper investigation on the dynamic view of software bugs across a wider range of software ties. In particular, we investigate time-lag bug correlations between software ties, design new ties (graph) exhibiting more severe bug correlation, and discuss the casual mechanisms behind this phenomenon.

7.2 Software Graphs

Besides our work, graph models have been widely used in empirical software engineering [7, 26, 58] especially for defect prediction [62, 63]. A typical use of graph models is in the analysis of developers' social networks [25, 55, 35]. Meneely et al. [37] leverage graphs to model collaborations among developers. They examine the graph metrics in a developers' collaboration network to predict failures. Pinzger et al. [45] modeled developer-modules as graphs. They found correlations between centrality measures of developer-module networks and failure-prone modules, and used such correlations to predict failures.

Dependency in graphs is also an active research topic. Zimmermann [62, 63] focused on the software dependency graph which considers interactions between software artifacts. They introduced both *ego networks* and *global networks*, using graphical metrics to enhance the performance of defect prediction. While most graph-based analysis studied social factors and program dependency information separately, Bird et al. [8] found that task assignment and dependency structure interact to influence the quality of the resulting software. Their prediction method combining social networks with dependency structures was able to predict failure proneness with better accuracy.

In addition to defect prediction, graphs are also widely used in other domains in Software Engineering such as the research of software evolution. To better observe the evolution of very large software systems, Pinzger et al. [44] proposed a visualization approach that can provide condensed graphical views on source codes and release history data for multiple releases. Bhattacharya [7] analyze the evolution of software using social network and graph metrics in several prediction tasks. Both these works made use of the dependence between software artifacts and leveraged the dependence as one of the metrics for defect prediction.

Despite the successful application of graph models in software engineering, these approaches often use macro-level graph metrics (e.g., degrees, centralities) directly without analyzing the defect correlations between graph nodes. Our study differs from these works by providing a deeper insight into micro-level bug correlation and propagation.

8 Conclusion

In this paper we applied correlation analysis from the social science literature to identify and measure the propagation of software bugs as a source of temporal correlations between software artifacts. We first investigated temporal bug correlation on known software ties such as calls, inheritance and co-changes. We performed longitudinal logistic regressions on the time-lag correlations between neighboring bugs on different artifact ties. Our empirical study with data from four open source projects showed that software bugs can be temporally

correlated among some known software ties. Based on our findings, we synthesize a hybrid graph (tie) which exhibits a higher degree of bug correlation.

All these findings shed light on new insights to software maintenance. In addition to metrics of individual artifacts, researchers can consider the factor of temporal bug correlation when designing the bug localization and prediction models.

In the future, we will investigate temporal bug correlations on more graphs. We will also apply the temporal bug correlation properties to improve related software engineering tasks such as defect prediction, correlated bug report linking, and bug localization. In particular, we will use the graph neural network (GNN) [31] on the proposed hybrid graph to learn the representation of bug correlations.

Our tool and experimental data described in this paper are available at `http://github.com/bugcorrelation/bugcorrelation`.

References

1 Understand,http://www.scitools.com/. URL: `http://www.scitools.com/`.

2 Wala project. http://wala.sourceforge.net/. URL: `http://wala.sourceforge.net/`.

3 Jun Ai, Wenzhu Su, Shaoxiong Zhang, and Yiwen Yang. A software network model for software structure and faults distribution analysis. *IEEE Transactions on Reliability*, 68(3):844–858, 2019.

4 Aris Anagnostopoulos, Ravi Kumar, and Mohammad Mahdian. Influence and correlation in social networks. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 7–15, New York, NY, USA, 2008. ACM. `doi:10.1145/1401890.1401897`.

5 Carina Andersson and Per Runeson. A replicated quantitative analysis of fault distributions in complex software systems. *Software Engineering, IEEE Transactions on*, 33(5):273–286, 2007.

6 Sinan Aral, Lev Muchnik, and Arun Sundararajan. Distinguishing influence-based contagion from homophily-driven diffusion in dynamic networks. *Proceedings of the National Academy of Sciences*, 106(51):21544–21549, 2009. `doi:10.1073/pnas.0908800106`.

7 Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 419–429, Piscataway, NJ, USA, 2012. IEEE Press. URL: `http://dl.acm.org/citation.cfm?id=2337223.2337273`.

8 C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *Software Reliability Engineering, 2009. ISSRE '09. 20th International Symposium on*, pages 109–119, November 2009. `doi:10.1109/ISSRE.2009.17`.

9 John T Cacioppo, James H Fowler, and Nicholas A Christakis. Alone in the crowd: the structure and spread of loneliness in a large social network. *Journal of personality and social psychology*, 97(6):977, 2009.

10 Peter J Carrington, John Scott, and Stanley Wasserman. *Models and methods in social network analysis*. Cambridge university press, 2005.

11 Yujun Chen, Xian Yang, Hang Dong, Xiaoting He, Hongyu Zhang, Qingwei Lin, Junjie Chen, Pu Zhao, Yu Kang, Feng Gao, Zhangwei Xu, and Dongmei Zhang. Identifying linked incidents in large-scale online service systems. In *Proceedings of the 2020 ESEC/FSE*. ACM, 2020.

12 Nicholas A Christakis and James H Fowler. The spread of obesity in a large social network over 32 years. *New England journal of medicine*, 357(4):370–379, 2007.

13 Nicholas A Christakis and James H Fowler. Social contagion theory: examining dynamic social networks and human behavior. *Statistics in medicine*, 32(4):556–577, 2013.

14 M. D'Ambros, M. Lanza, and R. Robbes. On the relationship between change coupling and software defects. In *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, pages 135–144, October 2009. `doi:10.1109/WCRE.2009.19`.

15 Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010.

16 S. Davies and M. Roper. Bug localisation through diverse sources of information. In *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 126–131, 2013.

17 Harpal Dhama. Quantitative models of cohesion and coupling in software. *Journal of Systems and Software*, 29(1):65–74, 1995.

18 Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1):5–24, 2017.

19 James H. Fowler, Nicholas A. Christakis, Steptoe, and Diez Roux. Dynamic spread of happiness in a large social network: Longitudinal analysis of the framingham heart study social network. *BMJ: British Medical Journal*, 338(7685):pp. 23–27, 2009. URL: `http://www.jstor.org/stable/20511686`.

20 Katerina Goseva-Popstojanova and Jacob Tyo. Identification of security related bug reports via text mining using supervised and unsupervised classification. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 344–355. IEEE, 2018.

21 Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on*, 26(7):653–661, 2000.

22 Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Tracking the software quality of android applications along their evolution (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 236–247. IEEE, 2015.

23 Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 392–401. IEEE Press, 2013.

24 Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 121–130. IEEE Press, 2013.

25 Qiaona Hong, Sunghun Kim, SC Cheung, and Christian Bird. Understanding a developer social network and its evolution. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 323–332. IEEE, 2011.

26 Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 111–120. ACM, 2009.

27 Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 279–289, November 2013. `doi:10.1109/ASE.2013.6693087`.

28 JM Juran and FM Gryna. Juranís quality control handbook. *NY: McGraw-Hill*, 1988.

29 David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 137–146, New York, NY, USA, 2003. ACM. `doi:10.1145/956750.956769`.

30 Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

31 Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

32 Akif Günes Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. An investigation into the functional form of the size-defect relationship for software modules. *Software Engineering, IEEE Transactions on*, 35(2):293–304, 2009.

33 An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 476–481. IEEE, 2015.

34 Kung-Yee Liang and Scott L Zeger. Longitudinal data analysis using generalized linear models. *Biometrika*, 73(1):13–22, 1986.

35 Wanwangying Ma, Lin Chen, Yibiao Yang, Yuming Zhou, and Baowen Xu. Empirical analysis of network measures for effort-aware fault-proneness prediction. *Information and Software Technology*, 69:50–70, 2016.

36 Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, pages 308–320, 1976.

37 Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 13–23, New York, NY, USA, 2008. ACM. doi:10.1145/1453101.1453106.

38 Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. Architecture anti-patterns: Automatically detectable violations of design principles. *IEEE Transactions on Software Engineering*, 2019.

39 Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. Bug propagation through code cloning: An empirical study. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 227–237. IEEE, 2017.

40 Alessandro Murgia, Giulio Concas, Michele Marchesi, Roberto Tonelli, and Ivana Turnu. An analysis of bug distribution in object oriented systems. *arXiv preprint arXiv:0905.3296*, 2009.

41 N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 364–373, September 2007. doi:10.1109/ESEM.2007.13.

42 Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, David Lo, and Chengnian Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 70–79. IEEE, 2012.

43 Zhen Ni, Bin Li, Xiaobing Sun, Tianhao Chen, Ben Tang, and Xinchen Shi. Analyzing bug fix for automatic bug cause classification. *Journal of Systems and Software*, 163:110538, 2020.

44 Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 67–75. ACM, 2005.

45 Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 2–12, New York, NY, USA, 2008. ACM. doi:10.1145/1453101.1453105.

46 Sarah J. Ratcliffe and Justine Shults. GEEQBOX: A matlab toolbox for generalized estimating equations and quasi-least squares. *Journal of Statistical Software*, 25(14):1–14, May 2008. URL: http://www.jstatsoft.org/v25/i14.

47 H. Sajnani, V. Saini, and C. V. Lopes. A comparative study of bug patterns in java cloned and non-cloned code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 21–30, 2014.

48 Gehan MK Selim, Liliane Barbour, Weiyi Shang, Bram Adams, Ahmed E Hassan, and Ying Zou. Studying the impact of clones on software defects. In *2010 17th Working Conference on Reverse Engineering*, pages 13–21. IEEE, 2010.

49 Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 253–262. IEEE, 2011.

50 Chenhao Tan, Jie Tang, Jimeng Sun, Quan Lin, and Fengjiao Wang. Social action tracking via noise tolerant time-varying factor graphs. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 1049–1058, New York, NY, USA, 2010. ACM. doi:10.1145/1835804.1835936.

51 Jie Tang, Jimeng Sun, Chi Wang, and Zi Yang. Social influence analysis in large-scale networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 807–816, New York, NY, USA, 2009. ACM. doi:10.1145/1557019.1557108.

52 C. Tantithamthavorn, A. Ihara, and K. Matsumoto. Using co-change histories to improve bug localization performance. In *2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 543–548, 2013.

53 Justin G Trogdon, James Nonnemaker, and Joanne Pais. Peer effects in adolescent overweight. *Journal of health economics*, 27(5):1388–1399, 2008.

54 Ye Wang, Na Meng, and Hao Zhong. An empirical study of multi-entity changes in real bug fixes. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 287–298. IEEE, 2018.

55 Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. Predicting build failures using social network analysis on developer communication. In *Proceedings of the 31st International Conference on Software Engineering*, pages 1–11. IEEE Computer Society, 2009.

56 Lu Xiao, Yuanfang Cai, and Rick Kazman. Design rule spaces: A new form of architecture insight. In *Proceedings of the 36th International Conference on Software Engineering*, pages 967–977, 2014.

57 Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 51–62. IEEE, 2016.

58 Marcelo Serrano Zanetti, Ingo Scholtes, Claudio Juan Tessone, and Frank Schweitzer. Categorizing bugs with social networks: A case study on four open source software communities. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1032–1041, Piscataway, NJ, USA, 2013. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=2486788.2486930.

59 Hongyu Zhang. On the distribution of software faults. *IEEE Transactions on Software Engineering*, pages 301–302, 2007.

60 J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 14–24, 2012.

61 Tom Zimmerman, Nachiappan Nagappan, Kim Herzig, Rahul Premraj, and Laurie Williams. An empirical study on the relation between dependency neighborhoods and failures. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 347–356. IEEE, 2011.

62 Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 531–540, New York, NY, USA, 2008. ACM. doi:10.1145/1368088.1368161.

63 Thomas Zimmermann and Nachiappan Nagappan. Predicting defects with program dependencies. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 435–438, Washington, DC, USA, 2009. IEEE Computer Society. doi:10.1109/ESEM.2009.5316024.

Type-Directed Operational Semantics for Gradual Typing

Wenjia Ye
The University of Hong Kong, Hong Kong

Bruno C. d. S. Oliveira
The University of Hong Kong, Hong Kong

Xuejing Huang
The University of Hong Kong, Hong Kong

Abstract

The semantics of gradually typed languages is typically given indirectly via an elaboration into a cast calculus. This contrasts with more conventional formulations of programming language semantics, where the semantics of a language is given directly using, for instance, an operational semantics.

This paper presents a new approach to give the semantics of gradually typed languages directly. We use a recently proposed variant of small-step operational semantics called *type-directed operational semantics* (TDOS). In TDOS type annotations become operationally relevant and can affect the result of a program. In the context of a gradually typed language, such type annotations are used to trigger type-based conversions on values. We illustrate how to employ TDOS on gradually typed languages using two calculi. The first calculus, called λB^g, is inspired by the semantics of the blame calculus, but it has implicit type conversions, enabling it to be used as a gradually typed language. The second calculus, called λB^r, explores a different design space in the semantics of gradually typed languages. It uses a so-called *blame recovery semantics*, which enables eliminating some false positives where blame is raised but normal computation could succeed. For both calculi, type safety is proved. Furthermore we show that the semantics of λB^g is sound with respect to the semantics of the blame calculus, and that λB^r comes with a *gradual guarantee*. All the results have been mechanically formalized in the Coq theorem prover.

2012 ACM Subject Classification Theory of computation → Type theory; Software and its engineering → Object oriented languages; Software and its engineering → Polymorphism

Keywords and phrases Gradual Typing, Type Systems, Operational Semantics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.12

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.9`

Funding This work has been sponsored by Hong Kong Research Grant Council projects number 17209519 and 17209520.

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

Gradual typing aims to provide a smooth integration between the static and dynamic typing disciplines. In gradual typing a program with no type annotations behaves as a dynamically typed program, whereas a fully annotated program behaves as a statically typed program. The interesting aspect of gradual typing is that programs can be partially typed in a spectrum ranging from fully dynamically typed into fully statically typed. Several mainstream languages, including TypeScript [6], Flow [11] or Dart [8] enable forms of gradual typing to various degrees. Much research on gradual typing has focused on the pursuit of sound gradual typing, where certain type safety properties, and other properties about the transition between dynamic and static typing, are preserved.

The semantics of gradually typed languages is typically given indirectly via an elaboration into a cast calculus. For instance the *blame calculus* [39, 52], the *threesome calculus* [42] or other cast calculi [15, 20, 23, 37, 39, 49] are often used to give the semantics of gradually typed languages. Since a gradual type system can accept programs with unknown types, run-time checks are necessary to ensure type safety. Thus the job of the (type-directed) elaboration is to insert casts that bridge the gap between known and unknown types. Then the semantics of a cast calculus can be given in a conventional manner.

While elaboration is the most common approach to give the semantics for gradually typed languages, it is also possible to have a direct semantics. In fact, a direct semantics is more conventionally used to provide the meaning to more traditional forms of calculi or programming languages. A direct semantics avoids the extra indirection of a target language and can simplify the understanding of the language. Garcia et al. [17], as part of their *Abstracting Gradual Typing* (AGT) approach, advocated and proposed an approach for giving a direct semantics to gradually typed languages. They showed that the cast insertion step provided by elaboration, which was until then seen as essential to gradual typing, could be omitted. Instead, in their approach, they develop the dynamic semantics as proof reductions over source language typing derivations.

This paper presents a different approach to give the semantics of gradually typed languages directly. We use a recently proposed variant of small-step operational semantics [54] called *type-directed operational semantics* (TDOS) [25]. For the most part developing a TDOS is similar to developing a standard small step-semantics as advocated by Wright and Felleisen. However, in TDOS type annotations become operationally relevant and can affect the result of a program. While there have been past formulations of small-step semantics where type annotations are also relevant [5, 14, 18], the distinctive feature of TDOS is a so-called *typed reduction* relation. Typed reduction further reduces values based on their types. While typically values are the final result of a program, in TDOS typed reduction can further transform them based on their run-time type. Thus typed reduction provides an operational interpretation to type conversions in the language, similarly to coercions in coercion-based calculi [23].

We illustrate how to employ TDOS on gradually typed languages using two calculi. The first calculus, called λB^g, is inspired by the semantics of a variant of the blame calculus (λB) [52] by Siek et al. [39]. However, unlike the blame calculus, λB^g allows implicit type conversions, enabling it to be used as a gradually typed language. Gradually typed languages can be built on top of λB using an elaboration from a source language into λB. In contrast λB^g can already act as a gradual language, without the need for an elaboration process.

The second calculus, called λB^r, explores a different design space in the semantics of gradually typed languages. It uses a so-called *blame recovery semantics*, which enables eliminating some false positives where blame is raised but normal computation could succeed. In the λB calculus, a lambda expression annotated with a chain of types is taken as a value. This means that it accumulates the type annotations, and checks if there are errors only when the function is applied to a value. This has some drawbacks. Perhaps most notably, and widely discussed in the literature [16, 24, 37, 38, 42], is that the accumulation of annotations affects space efficiency. Moreover, sometimes blame is raised quite conservatively, when a program could successfully return a value. Of course, the blame calculus semantics is justified by its origins on contracts and traditional casts (such as those commonly used in mainstream languages like Java). In such settings all casts/contracts must be valid, and any violations should raise blame.

In the λB^r calculus, the design choice that we make is to only raise blame if the initial source type of the value and final target types are not consistent. Otherwise, even if intermediate annotations trigger type conversions which would not be consistent, the final result can still be a value provided that the initial source and final target types are themselves consistent. This semantics differs from the blame calculus where intermediate types can cause blame. Technically speaking we introduce a new *saved expression/value*, that is used as an intermediate result during reduction. A saved value is generated whenever a conversion between two inconsistent types is triggered. However, if later another type conversion is applied to the value, then a saved value can recover from the brink of blame and be restored as a conventional value, provided that the new target type is consistent with the type of the saved value. A nice aspect of this semantics is that it avoids the accumulation of type annotations, being more space efficient.

For both calculi type safety is proved. Furthermore we show that the semantics of λB^g is sound with respect to the semantics of the blame calculus, and that λB^r comes with a *gradual guarantee* [41]. All the results have been mechanically formalized in the Coq theorem prover.

In summary, the contributions of this work are:

- **TDOS for gradual typing:** We show that TDOS can be employed in gradually typed languages. This enables simple, and concise specifications of the semantics of gradually typed languages, without resorting to an intermediate cast calculus. A nice aspect of TDOS is that it remains close to the simple and familiar small-step semantics approach by Wright and Felleisen.
- **The λB^g calculus** provides a first concrete illustration of TDOS for gradual typing. It follows closely the semantics of the blame calculus, but it allows implicit type conversions. We show type-safety, determinism, as well as a soundness theorem that relates the semantics of λB^g to that of the blame calculus (ignoring blame labels).
- **The λB^r calculus and blame recovery semantics.** λB^r explores the design space of the semantics of gradual typing by using a blame recovery semantics. The key idea is to only raise blame if the initial source type of the value and final target types are not consistent. Furthermore, λB^r comes with a gradual guarantee [41].
- **Coq Formalization:** Both λB^g and λB^r, and all associated lemmas and theorems, have been formalized in the Coq theorem prover. The Coq formalization can be found in the supplementary materials of this paper:

`https://github.com/YeWenjia/TypedDirectedGradualTyping`

2 Overview

This section provides background on gradual typing and the blame calculus, and then illustrates the key ideas of our work and the λB^g and λB^r calculi.

2.1 Background: Gradual Typing and the λB calculus

Traditionally, programming languages can be divided into statically typed languages and dynamically typed languages. For a statically typed language, the type of every term must be known. The language may support type inference, but it usually requires some type annotations by the programmer, which bears some extra work for a programmer. However, the benefit of static typing is that type-unsafe programs are rejected before they are executed. On the other hand, in dynamically typed languages terms do not have static types and no

type annotations are needed. This waives the burden of a strict type discipline, at the cost of type-safety.

Gradual typing [43] is like a bridge connecting the two styles. Gradual typing extends the type system of static languages by allowing terms to have a *dynamic type* $\star$, which stands for the possibility of being any type. A term with the unknown type $\star$ is not rejected in any context by the type checker. Therefore, it can be viewed as in a dynamically typed language. In a gradually typed language, programs can be completely statically typed, or completely dynamically typed, or anything in between.

To cooperate with the very flexible $\star$ type, the common practice in gradual type systems is to define a binary relation called type *consistency*. A term of type A can be assigned type B if A and B are consistent ($A \sim B$). With $\star$ defined to be consistent with any other type, dynamic snippets can be embedded into the whole program without breaking the type soundness property. Of course, the type soundness theorem is relaxed and tolerates some kinds of run-time type errors. Besides type soundness, there are some other criteria for gradual typing systems. One well-recognized standard is the gradual guarantee proposed by Siek et al. [41].

Elaboration semantics of Gradual Typing and the λB calculus. The semantics of gradually typed languages is usually given by an elaboration into a cast calculus. This approach has been widely used since the original work on gradual typing by both Siek and Taha [43] and Tobin-Hochstadt and Felleisen [49].

One of the most widely used cast calculus for the elaboration of gradually typed languages is the blame calculus [39, 52]. Figure 1 shows the definition of the blame calculus. Here we base ourselves in a variant of the blame calculus by Siek et al. [39], but ignoring blame labels. The blame calculus is the simply-typed lambda calculus extended with the dynamic type ($\star$) and the cast expression ($t : A \Rightarrow B$). Meta-variables G and H range over ground types, which include Int and $\star \to \star$. The definition of values in the blame calculus contains some interesting forms. In particular, casts ($V : A \to B \Rightarrow A' \to B'$) and $V : G \Rightarrow \star$ are included. Run-time type errors are denoted as *blame*. Besides the standard typing rules of the simply typed lambda calculus, there is an additional typing rule for casts: if term t has type A and A is consistent with B, a cast on t from A to B has type B. The consistency relation for types states that every type is consistent with itself, $\star$ is consistent with all types, and function types are consistent only when input types and output types are consistent with each other. In the premise of rule BSTEPP-DYNA, there is a function ug which says that type A should be a function type consistent with $\star \to \star$, but not $\star \to \star$ itself.

The bottom of Figure 1 shows the reduction rules that we use in this paper. The dynamic semantics of the λB calculus is standard for most rules. The semantics of casts include the noteworthy parts. For first-order values, reduction is straightforward: a cast either succeeds or it fails and raises blame. For example:

$$1 : Int \Rightarrow \star : \star \Rightarrow Int \longmapsto^* 1$$
$$1 : Int \Rightarrow \star : \star \Rightarrow Bool \longmapsto^* blame$$

For higher-order values such as functions, the semantics is more complex, since the casted result cannot be immediately obtained. For example, if we cast from $\star \to \star$ to $\mathsf{Int} \to \mathsf{Int}$, we cannot judge the cast result immediately. So the checking process is deferred until the function is applied to an argument. Rule BSTEPP-ABETA shows that process: a function with the cast is a value which does not reduce until it has been applied to a value.

Syntax

$$
\begin{array}{ll}
\textit{Types} & A, B ::= \mathit{Int} \mid \star \mid A \to B \\
\textit{Ground types} & G, H ::= \mathit{Int} \mid \star \to \star \\
\textit{constant} & c ::= i \mid ... \\
\textit{Terms} & t ::= c \mid x \mid t : A \Rightarrow B \mid t_1\ t_2 \mid \lambda x : A.t \\
\textit{Result} & r ::= t \mid \mathit{blame} \\
\textit{Values} & V, W ::= c \mid V : A \to B \Rightarrow A' \to B' \mid \lambda x : A.t \mid V : G \Rightarrow \star \\
\textit{Context} & \Gamma ::= \cdot \mid \Gamma, x : A \\
\textit{Frame} & F ::= []\ t \mid V\ [] \mid [] : A \Rightarrow B
\end{array}
$$

$\boxed{\Gamma \vdash t : A}$ *(Additional Typing Rules)*

$$\textsc{btyp-cast}\ \frac{\Gamma \vdash t : A \qquad A \sim B}{\Gamma \vdash t : A \Rightarrow B : B}$$

$\boxed{A \sim B}$ *(Consistency of types)*

$$\textsc{s-i}\ \frac{}{\mathsf{Int} \sim \mathsf{Int}} \qquad \textsc{s-arr}\ \frac{A \sim C \qquad B \sim D}{A \to B \sim C \to D} \qquad \textsc{s-dynl}\ \frac{}{\star \sim A} \qquad \textsc{s-dynr}\ \frac{}{A \sim \star}$$

$\boxed{t \longmapsto r}$ *(Reduction for the λB Calculus)*

$$\textsc{bStepp-eval}\ \frac{t \longmapsto t'}{F.t \longmapsto F.t'} \qquad \textsc{bStepp-blame}\ \frac{t \longmapsto \mathit{blame}}{F.t \longmapsto \mathit{blame}} \qquad \textsc{bStepp-beta}\ \frac{}{(\lambda x : A.\, t)\, V \longmapsto t[x \mapsto V]}$$

$$\textsc{bStepp-vany}\ \frac{}{(V : G \Rightarrow \star) : \star \Rightarrow G \longmapsto V} \qquad \textsc{bStepp-dd}\ \frac{}{V : \star \Rightarrow \star \longmapsto V}$$

$$\textsc{bStepp-dyna}\ \frac{ug(A, \star \to \star)}{V : \star \Rightarrow A \longmapsto (V : \star \Rightarrow \star \to \star) : \star \to \star \Rightarrow A} \qquad \textsc{bStepp-blamep}\ \frac{G \nsim H}{(V : G \Rightarrow \star) : \star \Rightarrow H \longmapsto \mathit{blame}}$$

$$\textsc{bStepp-abeta}\ \frac{}{(V : A \to B \Rightarrow A' \to B')\, V \longmapsto (V\, (V : A' \Rightarrow A)) : B \Rightarrow B'} \qquad \textsc{bStepp-lit}\ \frac{}{i : \mathsf{Int} \Rightarrow \mathsf{Int} \longmapsto i}$$

$$\textsc{bStepp-anyd}\ \frac{ug(A, \star \to \star)}{V : A \Rightarrow \star \longmapsto (V : A \Rightarrow \star \to \star) : \star \to \star \Rightarrow \star}$$

Figure 1 The λB Calculus (selected rules).

2.2 Motivation for a Direct Semantics for Gradual Typing

In this paper we propose not to use an elaboration semantics into a cast calculus, but to use a direct semantics for gradual typing instead. We are not the first to propose such an approach. For instance, the AGT framework for gradual typing [17] also employs a direct semantics. In that work the authors state that "*developing dynamic semantics for gradually typed languages has typically involved the design of an independent cast calculus that is peripherally related to the source language*". They further argue that there is a gap between source gradually typed languages, and the cast calculi that they target. In particular cast calculi admit "*far more programs than those in the image of the translation procedure*". We agree with such arguments. In addition, as argued by Huang and Oliveira [25], there are some other reasons why a direct semantics is beneficial over an elaboration semantics.

A direct semantics enables simple ways for programmers and tools to reason about the behaviour of programs. For instance, with languages like Haskell it is quite common for programmers to use equational reasoning. Such reasoning steps are directly justifiable from the operational semantics of call-by-name/need languages. With a TDOS, we can easily (and justifiably) employ similar steps to reason about your source language (say GTLC or λB^g). With a semantics defined via elaboration, however, that is not an easy thing because of the indirect semantics. We refer readers to Huang and Oliveira's work, which has an extensive discussion about this point. Additionally, some tools, especially some debuggers or tools for demonstrating how programs are computed, require a direct semantics, since those tools need to show transformations that happen after some evaluation of the *source program*.

Another potential benefit of a direct semantics is simpler and shorter metatheory/implementation. For instance, with a direct semantics we can often save quite a few definitions/proofs, including a second type system, various definitions on well-formedness of terms, substitution operations and lemmas, pretty printers, etc. Though these are not arguably difficult, they do add up. Perhaps more importantly, some proofs can be simpler with a direct semantics. For example, proving the gradual guarantee is typically simpler, since some lemmas that are required with an elaboration semantics (for example, Lemma 6 in original work on the refined criteria for gradual typing [41]) are not needed with a direct operational semantics. Moreover only the precision relation for the source language is necessary.

2.3 λB^g: A Gradually Typed Lambda Calculus

Since λB requires explicit casts whenever a term's type is converted, it cannot be considered a gradually typed calculus. For comparison, the application rule for typing in the *Gradually Typed Lambda Calculus* (GTLC) [38, 41, 43]

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_3 \qquad T_1 \sim T_3}{\Gamma \vdash e_1\, e_2 : T_2}\ \text{GTLC-App}$$

does not force the input term to have the same type as what the function expects. It just checks the compatibility of the two terms' types and can do implicit type conversions (casts) automatically. In a cast calculus, similar flexibility only exists when the term is wrapped with a cast, since the application rule strictly requires the argument type to be of the same type of the input of the function type. In λB, for instance, the application rule is the same as in the Simply Typed Lambda Calculus, requiring the argument type to be of the same type of the input of the function type.

Bi-directional type-checking for λB^g. As a first step to adapt a λB-like calculus into a source language for gradual typing, we turn to the bidirectional type checking [33]. Unlike in GTLC or λB, a bidirectional typing judgement may be in one of the two modes: inference or checking. In the former, a type is synthesized from the term. In the later, both the type and the term are given as input, and the typing derivation examines whether the term can be used under that type safely. In a typical bidirectional type system with subtyping, the subsumption rule is only employed in the checking mode, allowing a term to be checked by a supertype of its inferred type. That is to say, the checking mode is more relaxed than the inference mode, which typically infers a unique type. With bidirectional type-checking the application rule in such a system is not as strict as in the λB calculus, as the input term is typed with a checking mode.

Implicit type conversion in function applications. By using bidirectional type checking, we can type-check programs such as:

$(\lambda x.x)\ 1$ Accepted!

$(\lambda x.not\ x)\ 1$ Accepted!

$(\lambda x.not\ x : \star \to Bool)\ 1$ Accepted!

$(\lambda x.x + 1 : Int \to Int)\ 1$ Accepted!

and also reject ill-typed programs:

$(\lambda x.not\ x : Bool \to Bool)\ 1$ Rejected!

Note that λB^g supports *annotation expressions* of the form $e : A$. Thus, an expression like $\lambda x.not\ x : Bool \to Bool$ is a lambda expression $(\lambda x.not\ x)$ annotated with the type $Bool \to Bool$.

Explicit type conversion. Besides implicit conversions, programmers are able to trigger type conversions in an explicit fashion by wrapping the term with a type annotation $e : A$, where A denotes the target type. For instance, the two simple examples in λB in Section 2.1 can be encoded in λB^g as:

$1 : \star : Int \longmapsto^* 1$

$1 : \star : Bool \longmapsto^* blame$

with similar results to the same programs in the λB calculus. Notice that, unlike λB, there is no cast expression in λB^g. Casts are triggered by type annotations. For instance, in the first expression above $(1 : \star : Int)$, the first type annotation $(\star)$ triggers a cast from Int to $\star$. The source type Int is the type of 1, whereas the target type $\star$ is specified by the annotation. Then the second annotation Int will trigger a second cast, but now from $\star$ to Int.

Functions. One interesting change in the type system is that we handle lambdas by inference mode rather than checking mode. Our rule for lambdas is:

$$\frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e : A \to B \Rightarrow A \to B}\ \textsc{Typ-abs}$$

If the programmer wants to have their function statically type checked, they can write down the full annotations. Otherwise, the function can be left with no annotation, which will be desugared into a lambda with type $\star \to \star$, similarly to what happens in the GTLC for dynamically typed lambdas.

2.4 Designing a TDOS for λB^g

The most interesting aspect of λB^g is its dynamic semantics. We discuss the key ideas next.

Background: Type-Directed Operational Semantics. A type-directed operational semantics is helpful for language features whose semantics is type dependent. TDOS was originally proposed for languages that have intersection types and a merge operator [25]. To enable expressive forms of the merge operator the dynamic semantics has to account for the types, just like the semantics of gradually typed languages. In many traditional operational semantics type annotations are often ignored. In TDOS that is not the case, and the type annotations are used at runtime to determine the result of reduction. A TDOS has two parts. One part is similar to the traditional reduction rules, modulo some changes on type-related rules, like beta reduction for application, and annotation elimination for values. The second component of a TDOS is the typed reduction relation $v \longmapsto_A r$. Typed reduction has a value and a type as input and produces a value (when no run-time error is possible) as result. The resulting value is transformed from the input value.

Typed Reduction for λB^g. Due to consistency, run-time checking is needed in gradual typing. The typed reduction relation $v \longmapsto_A r$ is used when run-time checks are needed. Typed reduction compares the dynamic type of the input value with the target type. When the type of the input value (v) is not consistent to the target type (A), blame is raised. Otherwise, typed reduction adapts the value to suit the target type. Eventually, terms become more and more precise. Two easy examples to show how typed reduction work are shown next:

$$1 \longmapsto_{Int} 1$$

$$1 : \star \longmapsto_{Bool} \mathit{blame}$$

If we have an integer value 1 and we want to transform it with type Int, we simply return the original value. In contrast, attempting to transform the value $1 : \star$ under type $Bool$ will result in blame.

Typed reduction takes place in other reduction rules such as the beta reduction rule and the annotation elimination rule for values:

$$\textsc{Step-beta}\ \frac{v \longmapsto_A v'}{(\lambda x.\, e : A \to B)\, v \longmapsto e[x \mapsto v'] : B} \qquad \textsc{Step-annov}\ \frac{\mathit{not}\ (\mathit{value}\ (v : A)) \quad v \longmapsto_A r}{v : A \longmapsto r}$$

Take another example to illustrate the behavior of typed reduction in beta reduction:

$$(\lambda x.x : Bool \to Bool)\ (1 : \star)$$

If we would perform substitution directly, as conventionally done in beta-reduction, we would not check if there are run-time errors, for which blame should be raised. Since the typing rule for the argument of application is in checking mode, we need to check if the type of the argument is consistent with the target type. Therefore the argument must be further reduced with typed reduction under the expected type of the function input. When we check that the type Int is not consistent with $Bool$, blame is raised. However, if we take the example:

$$(\lambda x.x + 1 : Int \to Int)\ (1 : \star)$$

then the value 1 is substituted in the function body and the result is 2. The details of reduction and typed reduction in λB^g will be discussed in Section 3.

2.5 λB^r: Gradual Typing with a Blame Recovery Semantics

An alternative semantics for Gradual Typing. In λB^r we explore an alternative semantics for gradual typing that we call *blame recovery semantics*. The main idea is to only raise blame when the initial (source) type and the final target types in a chain of type annotations are inconsistent. Intermediate inconsistent types will not lead to blame. Thus the blame recovery semantics can be viewed as being more liberal with respect to raising blame. We illustrate the difference next, with 2 programs that raise blame in λB^g, but would successfully compute a value in λB^r:

$$(\lambda x.x : Int \to Int : \star \to \star : Bool \to Bool : \star \to \star)\ 1 \longmapsto^* blame \quad \{\text{Examples in } \lambda B^g\ \}$$
$$(\lambda x.x : \star \to \star : Bool \to Bool : \star \to \star : Int \to Int)\ 1 \longmapsto^* blame$$

$$(\lambda x.x : Int \to Int : \star \to \star : Bool \to Bool : \star \to \star)\ 1 \longmapsto^* 1 : Int \quad \{\text{Same examples in } \lambda B^r\ \}$$
$$(\lambda x.2 : \star \to \star : Bool \to Bool : \star \to \star : Int \to Int)\ 1 \longmapsto^* 2 : Int$$

For the above two examples, the function being applied is wrapped on a chain of annotations that contain the inconsistent types $\mathsf{Int} \to \mathsf{Int}$ and $Bool \to Bool$. Therefore, in λB^g blame is raised in both cases. However, in λB^r, because $\star \to \star$ is consistent with $\mathsf{Int} \to \mathsf{Int}$, the annotation chain of functions is eliminated, then beta reduction applies, and it successfully reduces to an integer value.

Space Efficiency. Besides the different semantics with respect to the λB^g and λB calculi, an interesting aspect of this alternative semantics is better space efficiency. Unlike the blame calculus or λB^g, where functions with an arbitrary number of annotations are values, that is not the case in λB^r. Because of the blame recovery semantics it is possible to discard intermediate types when reducing expressions with chains of annotations. Instead of wrappers for higher-order casts, function values have just 2 annotations. Some concrete examples for higher-order casts (functions) are:

$$(\lambda x.\ x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \mathsf{Int} \to \mathsf{Int} \longmapsto^* (\lambda x.\ x : \mathsf{Int} \to \mathsf{Int}) : \mathsf{Int} \to \mathsf{Int}$$
$$(\lambda x.\ x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : Bool \to Bool \longmapsto^* [\lambda x.x : Int \to Int]_{Bool \to Bool}$$
$$(\lambda x.\ x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : Bool \to Bool : \star \to \star \longmapsto^* \lambda x.\ x : \mathsf{Int} \to \mathsf{Int} : \star \to \star$$

For the first example, because the source type $\mathsf{Int} \to \mathsf{Int}$ is consistent with the target type $\mathsf{Int} \to \mathsf{Int}$, the intermediate types are ignored and the resulting value is $(\lambda x.\ x : \mathsf{Int} \to \mathsf{Int}) : \mathsf{Int} \to \mathsf{Int}$. For the second example, because the source type $\mathsf{Int} \to \mathsf{Int}$ is not consistent with the target type $Bool \to Bool$, instead of raising blame immediately, the source type will be stored in a saved value: $[\lambda x.\ x : \mathsf{Int} \to \mathsf{Int}]_{Bool \to Bool}$. Later, if the saved value is applied to an argument, blame will be raised, since a saved value is denoting that the function has inconsistent source and target types. The third example, is similar to the second example except that there is an extra final target type $\star \to \star$. Thus, since the initial source type $\mathsf{Int} \to \mathsf{Int}$ is consistent with the final target type $\star \to \star$ the final value is $(\lambda x.\ x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star$. The above examples illustrate that at most there will be 2 type annotations in values. In contrast, for the blame calculus, the three examples are values where all the annotations are accumulated.

Saved Expressions. To realize the blame recovery semantics we introduce *saved expressions/values*, which are used to signal *potential blame*. A saved value is generated whenever

some target type arising from an annotation is inconsistent with the current value. If further annotations are processed after a saved value is generated, recovery from blame is possible. Take the third example above again. The full reduction steps for that example are:

$$
\begin{aligned}
&(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \mathit{Bool} \to \mathit{Bool} : \star \to \star \\
&\longmapsto [\lambda x.x : \mathit{Int} \to \mathit{Int}]_{\mathit{Bool} \to \mathit{Bool}} : \star \to \star \\
&\longmapsto \lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \star \to \star
\end{aligned}
$$

An intermediate saved value is generated, but because there is still one more consistent annotation ($\star \to \star$), the value is recovered from the saved expression, revoking the potential reason to raise blame.

Blame in λB and λB^r. While λB^r has a different semantics from λB (and λB^g), the semantics of the two calculi is still closely related. In particular, with respect to blame, a program that does not raise blame in λB^g or λB will also not raise blame in λB^r. Achieving this goal is not simple, because of the semantics of the blame calculus and λB^g for higher-order casts. For instance, consider the following λB^g program:

$$((\lambda x.\, x : (\mathit{Bool} \to \mathit{Bool}) \to (\mathit{Bool} \to \mathit{Bool})) : \star \to \star : (\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int}))\, (\lambda x.\, x : \mathsf{Int} \to \mathsf{Int})$$

In this well-typed program the lambda expression being applied has inconsistent type annotations. However, because in the blame calculus and λB^g the semantics of higher-order casts is lazy, this program will *not raise blame*. Instead, it eventually reduces to:

$$(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \mathsf{Int} \to \mathsf{Int} : \star \to \star : \mathit{Bool} \to \mathit{Bool} : \mathit{Bool} \to \mathit{Bool} : \star \to \star : \mathsf{Int} \to \mathsf{Int}$$

which is another lambda expression (arising from the argument) with inconsistent type annotations.

In λB^r the applied lambda expression in the original program would first be reduced to:

$$[\lambda x.\, x : (\mathit{Bool} \to \mathit{Bool}) \to (\mathit{Bool} \to \mathit{Bool})]_{(\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int})}$$

To achieve the same semantics as λB^g or λB, the design of λB^r has to include rules that can still perform beta-reduction for saved values being applied. In particular, λB^r has the following rule :

$$\frac{v \;\longmapsto_{A_1, \star, A_2}\; v'}{([\lambda x.\, e : A_1 \to B_1]_{A_2 \to B_2})\, v \;\longmapsto\; e[x \mapsto v'] : B_1 : \star : B_2} \textsc{VStep-apps}$$

With such a rule, the program above is reduced in λB^r as follows:

$$
\begin{aligned}
&[\lambda x.x : (\mathit{Bool} \to \mathit{Bool}) \to (\mathit{Bool} \to \mathit{Bool})]_{(\mathit{Int} \to \mathit{Int}) \to (\mathit{Int} \to \mathit{Int})} (\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) \\
&\longmapsto^* (\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \mathsf{Int} \to \mathsf{Int}
\end{aligned}
$$

3 The λB^g Calculus: Syntax, Typing and Semantics

In this section, we will introduce the gradually typed λB^g calculus. The semantics of the λB^g calculus follows closely the semantics of the λB cast calculus, and it employs a type-directed operational semantics [25] to have a direct operational semantics. λB^g uses bidirectional type-checking [33]. We prove a soundness result between the semantics of the λB^g and λB calculi (ignoring blame labels), as well as the usual type soundness property.

Syntax

Types	$A, B ::= Int \mid \star \mid A \to B$
Ground types	$G ::= Int \mid \star \to \star$
Constants	$c ::= i \mid ...$
Terms	$e ::= c \mid x \mid e : A \mid e_1\ e_2 \mid \lambda x.e : A \to B$
Result	$r ::= e \mid blame$
Values	$v ::= c \mid v : A \to B \mid \lambda x.e : A \to B \mid v : \star$
Context	$\Gamma ::= \cdot \mid \Gamma, x : A$
Frame	$F ::= [] \ e \mid v \ [] \mid [] : A$
Typing modes	$\Leftrightarrow ::= \Rightarrow \mid \Leftarrow$
Syntactic sugar	$\lambda x.e \equiv \lambda x.e : \star \to \star$

value e *(Well-formed values for λB^g calculus)*

$$\frac{}{value\ c}\ \text{VALUE-C} \qquad \frac{}{value\ \lambda x.\, e : A \to B}\ \text{VALUE-ANNO} \qquad \frac{\rceil v \lceil = C \to D}{value\ v : A \to B}\ \text{VALUE-FANNO} \qquad \frac{Ground\ \rceil v \lceil}{value\ v : \star}\ \text{VALUE-DYN}$$

Figure 2 Syntax and well-formed values for the λB^g calculus.

3.1 Syntax

The syntax of λB^g calculus is shown in Figure 2.

Types and Ground types. Meta-variables A and B range over types. There is a basic type: the integer type Int. The calculus also has function types $A \to B$, and dynamic types $\star$. The type $\star$ is used to denote the dynamic type which is unknown. Just like in λB calculus, ground types include Int and $\star \to \star$.

Constants, Expressions and Results. Meta-variable c ranges over constants. Each constant is assigned a unique type. The constants include integers (i) of type Int. Expressions range over by the meta-variable e. There are some standard constructs which include: constants (c);variables (x); annotated expressions ($e : A$); application expressions ($e_1 e_2$) and lambda abstractions ($\lambda x.e : A \to B$). Note that lambda abstractions have the function type annotation $A \to B$, meaning that the input type is A and the output type is B. Similarly to GTLC, lambdas without type annotations are just sugar for a lambda with the annotation $\star \to \star$. Results (r) include all expressions and blame, which is used to denote cast-errors at run-time. Finally, note that, in our Coq formalization, constants such as addition of integers are implemented, but omitted here for simplicity of presentation.

Value and Contexts. The meta-variable v ranges over values. Values include constants (c); lambda abstractions ($\lambda x.e : A \to B$) and a special value with the syntax $v : A \to B$. Note that, similarly to λB, not all syntactic values are well-formed values. The *value* predicate, at the bottom of Figure 2, defines well-formed values. Lambda expressions annotated with a function type are values (rule VALUE-ANNO). A value v with a function type annotation is

a value if the dynamic type of the value is also a function type (rule VALUE-FANNO). The expression of $v : \star$ is a value only when the type of v is a ground type (rule VALUE-DYN). Constants are also values. Note that $\rceil v \lceil$ denotes the dynamic type of a value, and is defined as:

▶ **Definition 1** (Dynamic type). *$\rceil v \lceil$ denotes the dynamic type of the value v.*

$$\begin{aligned} \rceil i \lceil &= \mathsf{Int} \\ \rceil \lambda x.\, e : A \to B \lceil &= A \to B \\ \rceil v : A \lceil &= A \end{aligned}$$

The dynamic type is the most specific type of a value among all the other types. Finally, typing contexts are standard. Γ is used to track the bound variables x with their type A.

Frame and Typing modes. The meta-variable F ranges over frames [39] which is a form of evaluation contexts [31]. The frame is mostly standard, though it is perhaps noteworthy that it includes annotated expressions. $\Leftrightarrow$ is used to represent the two modes of the bidirectional typing judgment. The $\Rightarrow$ mode is the synthesis (inference) mode and $\Leftarrow$ mode is the checking mode.

3.2 Typing

We use bidirectional typing for our typing rules. The typing judgment is represented as $\Gamma \vdash e \Leftrightarrow A$, which means that the expression e could be inferred or checked by the type A under the typing environment Γ. We ignore the highlighted parts, and explain them later in Section 3.4.

Typing Relation. The typing relation of the λB^g calculus is shown in Figure 3. Most of the rules in inference mode follow the λB calculus's type system. The typing for constants (rule TYP-C) recovers the type of the constants using the definition of dynamic types. The rule TYP-VAR for variables is standard. For lambda expressions, the λB^g calculus is different from the λB calculus: in the λB^g calculus the type of a lambda expression is given. Thus the body of the lambda expression is checked with a target type that should be consistent to the type of the lambda body. For applications $e_1\ e_2$, the rule is standard for bi-directional type-checking: the type of e_1 is inferred, and the type of e_2 is checked against the domain type of e_1. The rule for annotations (rule TYP-ANNO) is also standard, inferring the annotated type, while checking the expression in the against the annotated type. All the consistency checks happen in the subsumption rule (rule TYP-SIM). However, it is important to notice that since the subsumption rule is in checking mode, all consistency checks can only happen when typing is invoked in the checking mode.

Two important properties of the typing relation is that it computes dynamic types for the inference mode, and if an expression e can be checked with type A, then e can be inferred with some type B:

▶ **Lemma 2** (Dynamic Types). *For any value v, if $\cdot \vdash v \Rightarrow A$ then $\rceil v \lceil = A$.*

▶ **Lemma 3** (Checking to inference mode). *If $\Gamma \vdash e \Leftarrow A$ then $\exists B$, $\Gamma \vdash e \Rightarrow B$.*

$\boxed{\Gamma \vdash e \Leftrightarrow A \rightsquigarrow \mathrm{t}}$ *(Typing of λB^g)*

$$\textsc{Typ-c}\ \frac{}{\Gamma \vdash c \Rightarrow \rceil c \lceil \rightsquigarrow \mathrm{c}} \qquad \textsc{Typ-var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A \rightsquigarrow x} \qquad \textsc{Typ-abs}\ \frac{\Gamma, x : A \vdash e \Leftarrow B \rightsquigarrow \mathrm{t}}{\Gamma \vdash \lambda x.\, e : A \to B \Rightarrow A \to B \rightsquigarrow \lambda x{:}A.\, t}$$

$$\textsc{Typ-app}\ \frac{\Gamma \vdash e_1 \Rightarrow A \to B \rightsquigarrow t_1 \quad \Gamma \vdash e_2 \Leftarrow A \rightsquigarrow t_2}{\Gamma \vdash e_1\, e_2 \Rightarrow B \rightsquigarrow t_1\ t_2} \qquad \textsc{Typ-anno}\ \frac{\Gamma \vdash e \Leftarrow A \rightsquigarrow \mathrm{t}}{\Gamma \vdash e : A \Rightarrow A \rightsquigarrow \mathrm{t}} \qquad \textsc{Typ-sim}\ \frac{\Gamma \vdash e \Rightarrow A \rightsquigarrow \mathrm{t} \quad A \sim B}{\Gamma \vdash e \Leftarrow B \rightsquigarrow t : A \Rightarrow B}$$

Figure 3 Type system of the λB^g calculus.

$\boxed{v \longmapsto_A r}$ *(Typed Reduction for λB^g calculus)*

$$\textsc{TReduce-abs}\ \frac{\rceil v \lceil = C \to D \quad C \to D \sim A \to B}{v \longmapsto_{A \to B} v : A \to B} \qquad \textsc{TReduce-v}\ \frac{\mathit{Ground}\ \rceil v \lceil}{v \longmapsto_\star v : \star} \qquad \textsc{TReduce-lit}\ \frac{}{i \longmapsto_{\mathsf{Int}} i} \qquad \textsc{TReduce-dd}\ \frac{}{v : \star \longmapsto_\star v : \star}$$

$$\textsc{TReduce-anyd}\ \frac{\mathit{FLike}\ \rceil v \lceil}{v \longmapsto_\star v : \star \to \star : \star} \qquad \textsc{TReduce-dyna}\ \frac{\mathit{FLike}\ A \qquad \rceil v \lceil \sim A}{v : \star \longmapsto_A v : A} \qquad \textsc{TReduce-vany}\ \frac{}{v : \star \longmapsto_{\rceil v \lceil} v} \qquad \textsc{TReduce-blame}\ \frac{\rceil v \lceil \nsim A}{v : \star \longmapsto_A \mathit{blame}}$$

Figure 4 Typed Reduction for the λB^g calculus.

Consistency. Consistency plays an important role in a gradually type lambda calculus. Consistency acts as a relaxed equality relation. The consistency relation is the same as λB, and is already shown in Figure 1. In consistency, the reflexivity and symmetry properties hold. However, it is well-known that consistency is not a transitive relation. If consistency were transitive then every type would be consistent with any other type [43].

3.3 Dynamic Semantics

The dynamic semantics of λB^g employs a type-directed operational semantics (TDOS) [25]. In TDOS, besides the usual reduction relation, there is a special *typed reduction* relation for values that is used to further reduce values based on the type of the value. Typed reduction is used by the TDOS reduction relation. In a gradually typed calculus with TDOS the typed reduction relation plays a role analogous to various cast-related reduction rules in a cast calculus. We first introduce typed reduction and then move on to the definition of reduction.

Typed Reduction. We reduce a value under a certain type using the typed reduction relation. The form of the typed reduction relation is $v \longmapsto_A r$, which means that a value v annotated with A reduces under type A to a result r. Note that the result r produced by typed reduction can only be a value or *blame*. Blame is raised during typed reduction if

we try to reduce the value under a type that is not consistent with the type of the value. For instance trying to reduce the value $1 : \star$ under the type $Bool$ will raise blame. Thus, it should be clear that typed reduction mimics the behavior of casts in cast calculi like the λB calculus. In the λB calculus, in a cast $t : B \Rightarrow A$, t should be a cast from a source type B to a target type A. Using typed reduction, the type A is the target type, whereas the dynamic type of v is the source type.

Figure 4 shows the rules of typed reduction. Rule TReduce-abs and rule TReduce-v just add a type annotation to the value. In rule TReduce-abs the dynamic type of the value is a function type, thus v annotated with $A \to B$ is a value. In rule TReduce-v, $v : \star$ is also a value when the dynamic type of v is a ground type. Rule TReduce-lit is for integer values: an integer i being reduced under the integer type results in the same integer i. A value $v : \star$ type-reduced under $\star$ returns the original value as well (rule TReduce-dd). In rule TReduce-anyd, the premise is that the dynamic type of v should be a function-like type ($FLike$). The definition of $FLike$, which plays a role analogous to $ug(A, \star \to \star)$ in λB, is:

$$\boxed{\text{FLike} \quad \text{A} \quad ::= \quad A \neq \star \wedge A \neq \star \to \star \wedge A \sim \star \to \star}$$

If a type A is $FLike$ then it is not the type $\star$ and the type $\star \to \star$, but should be consistent with $\star \to \star$. In other words, the dynamic type of v should be any function type $A \to B$ except for $\star \to \star$. In the end v is type-reduced under type $\star$ and returns the value $v : \star \to \star : \star$. In rule TReduce-vany, $v : \star$ is type-reduced under the dynamic type of v, returning v and dropping the annotation $\star$. In rule TReduce-blame, if the dynamic type of v is not consistent to the type A that we are type-reducing, then blame is raised. Finally, in rule TReduce-dyna, a value $v : \star$ being type-reduced under type A (where A is function-like and the dynamic type of v is consistent with A) results in $v : A$. That is the annotation $\star$ gets replaced by the function type A.

Properties of Typed Reduction. Some properties of typed reduction of λB^g calculus are shown next:

▶ **Lemma 4** (Typed reduction preserves well-formedness of values). *If value v and $v \longmapsto_A v'$ then value v'.*

▶ **Lemma 5** (Preservation of Typed Reduction). *If $\cdot \vdash v \Leftarrow B$ and $v \longmapsto_A v'$ then $\cdot \vdash v' \Rightarrow A$.*

▶ **Lemma 6** (Progress of Typed Reduction). *If $\cdot \vdash v \Leftarrow A$ then $\exists v'$, $v \longmapsto_A v'$ or $v \longmapsto_A$ blame.*

▶ **Lemma 7** (Determinism of Typed Reduction). *If $\cdot \vdash v \Leftarrow B$, $v \longmapsto_A r_1$ and $v \longmapsto_A r_2$ then $r_1 = r_2$.*

▶ **Lemma 8** (Typed Reduction Respects Consistency). *If $v \longmapsto_A v'$ then $\rceil v \lceil \sim A$.*

According to Lemma 4, if the result of a value type-reduced under a type A is not blame, then it should be a well-formed value. Lemma 5 shows that the target type A is preserved after typed reduction: if a value v is type-reduced by A, the result type of v' is of type A. Note that this lemma (and some others) have a premise that ensures that the value under typed reduction must be well-typed under some type B. That is, the lemma only holds for well-typed values (which are the only ones that we care about). Lemma 6 shows that if a value v is well-typed with A, then type-reducing the value will either return a

$\boxed{e \longmapsto r}$ *(Small-step Semantics)*

$$\textsc{Step-eval}\ \frac{e \longmapsto e'}{F.\,e \longmapsto F.\,e'} \qquad \textsc{Step-blame}\ \frac{e \longmapsto \mathit{blame}}{F.\,e \longmapsto \mathit{blame}} \qquad \textsc{Step-beta}\ \frac{v \longmapsto_A v'}{(\lambda x.\,e : A \to B)\,v \longmapsto e[x \mapsto v'] : B}$$

$$\textsc{Step-betap}\ \frac{v \longmapsto_A \mathit{blame}}{(\lambda x.\,e : A \to B)\,v \longmapsto \mathit{blame}} \qquad \textsc{Step-annov}\ \frac{\mathit{not}\ (\mathit{value}\ (v : A)) \quad v \longmapsto_A r}{v : A \longmapsto r}$$

$$\textsc{Step-abeta}\ \frac{\mathit{value}\ (v_1 : A \to B) \quad v_2 \longmapsto_A v_2'}{(v_1 : A \to B)\,v_2 \longmapsto (v_1\,v_2') : B} \qquad \textsc{Step-abetap}\ \frac{\mathit{value}\ (v_1 : A \to B) \quad v_2 \longmapsto_A \mathit{blame}}{(v_1 : A \to B)\,v_2 \longmapsto \mathit{blame}}$$

Figure 5 Semantics of λB^g.

well-formed value or blame. The typed reduction relation is deterministic for well-typed values (Lemma 7): if a well-typed value v is type-reduced by type A, the result will be unique. Finally, if v is type-reduced by A, the dynamic type of v should be consistent with type A (Lemma 8). Most of these lemmas are proved by induction on typed reduction relation.

Reduction. The reduction rules are shown in Figure 5. Rule Step-eval and rule Step-blame are standard evaluation context reduction rules. Rule Step-beta is the beta reduction rule. Importantly, note that typed reduction under type A is needed for v: that is we type-reduce value v to v' and replace the bound variable x in e by v'. Rule Step-betap applies when v type-reducing under type A raises blame. Rule Step-annov states that v type-reduces under type A to return r. Rule Step-abeta says v_2 type-reduces by type A to get v_2' and v_1 will erase the annotation. The expression $v_1\ v_2'$ in the result is annotated with type B. Rule Step-abetap covers the case when v_2 type-reducing under type A raises blame.

Determinism. The operational semantics of λB^g is *deterministic*: a well-typed expression reduces to a unique result. Theorem 9 is proved using Lemma 7.

▶ **Theorem 9** (Determinism of λB^g calculus). *If* $\cdot \vdash e \Leftarrow A$, $e \longmapsto r_1$ *and* $e \longmapsto r_2$ *then* $r_1 = r_2$.

Type Safety. The λB^g calculus is type safe. Theorem 10 says that if an expression is well-typed with type A, the type will be preserved after the reduction. Progress is given by Theorem 11. A well-typed expression e is either a value or there exists an expression e' which e could reduce to, or e reduces to blame.

▶ **Theorem 10** (Type Preservation of λB^g Calculus). *If* $\cdot \vdash e \Leftrightarrow A$ *and* $e \longmapsto e'$ *then* $\cdot \vdash e' \Leftrightarrow A$.

▶ **Theorem 11** (Progress of λB^g Calculus). *If* $\cdot \vdash e \Leftrightarrow A$ *then* e *is a value or* $\exists e'$, $e \longmapsto e'$ *or* $e \longmapsto \mathit{blame}$.

3.4 Soundness to λB

The judgment $\Gamma \vdash e \Leftrightarrow A \rightsquigarrow t$, shown in Figure 3 has an elaboration step from λB^g expressions to λB expressions in the gray portion of the judgement. This elaboration step is used to prove a soundness result between the semantics of λB^g and λB. A first property, given by Theorem 12, is that the elaboration is type-safe. Theorem 13 and Theorem 14 show the soundness property between the dynamic semantics of λB^g and λB. The soundness result is proved using the auxiliary lemmas 15 and 16.

▶ **Theorem 12** (Type-Safety of Elaboration). *If* $\Gamma \vdash e \Leftrightarrow A \rightsquigarrow t$ *then* $\Gamma \vdash t : A$.

▶ **Theorem 13** (Soundness of λB^g calculus semantics with respect to λB calculus semantics). *If* $\cdot \vdash e \Leftrightarrow A \rightsquigarrow t$ *and* $e \longmapsto e'$ *then* $\exists t'$, $t \longmapsto^* t'$ *and* $\cdot \vdash e' \Leftrightarrow A \rightsquigarrow t'$.

▶ **Theorem 14** (Soundness of λB^g calculus semantics with respect to λB calculus semantics). *If* $\cdot \vdash e \Leftrightarrow A \rightsquigarrow t$ *and* $e \longmapsto \mathit{blame}$ *then* $t \longmapsto^* \mathit{blame}$.

▶ **Lemma 15** (Soundness of Typed Reduction λB^g calculus with respect to λB calculus semantics). *If* $\cdot \vdash v : A \Rightarrow A \rightsquigarrow t$ *and* $v \longmapsto_A v'$ *then* $\exists t'$, $t \longmapsto^* t'$ *and* $\cdot \vdash v' \Rightarrow A \rightsquigarrow t'$.

▶ **Lemma 16** (Soundness of Typed Reduction λB^g calculus with respect to λB calculus semantics). *If* $\cdot \vdash v : A \Rightarrow A \rightsquigarrow t$ *and* $v \longmapsto_A \mathit{blame}$ *then* $t \longmapsto^* \mathit{blame}$.

4 The λB^r Calculus and the Blame Recovery Semantics

In this section, we will introduce a gradually typed calculus with a blame recovery semantics. The idea of the blame recovery semantics is essentially to ignore intermediate inconsistent types in annotations. Thus, if blame arises from intermediate type annotations, but later the final source type is found to be consistent to the final target type then blame is not raised. A nice aspect of the blame recovery semantics is that it avoids accumulating type annotations, leading to a more space-efficient representation of values. The details of syntax, typing and semantics of λB^r calculus are shown below.

4.1 Syntax

The syntax of the λB^r calculus is shown in Figure 6.

Types. Types are the same as in λB^g. A type is either a integer type Int, a function type $A \to B$ or a dynamic type $\star$.

Expressions and Results. For expressions and results, λB^r extends λB^g with two expression forms: base expressions and saved expressions. Base expressions (ss) include annotated lambda expressions and integers (i). Saved expressions ($[s]_{A \to B}$) store a lambda expression and a type $A \to B$ which is not consistent with the type of the lambda expression. The lambda expressions stored in saved expressions are denoted as s. In our Coq formalization, addition is also implemented and omitted here for simplicity of presentation.

Values. As in λB^g, v denotes values, which are base expressions ss or saved forms annotated with a type. Thus $i : A$ and $\lambda x.\, e : A \to B : C$ are examples of such expressions. Notably, in contrast with λB^g, λB^r's notion of (well-formed) values is purely syntactic: no additional constraints (besides) syntax are needed. Moreover, it should be noted that in λB^r values have a bounded number of annotations (up-to 2 for lambda and saved values), unlike the λB^g calculus.

Types	$A, B, C ::= Int \mid A \to B \mid \star$
Saved Forms	$s ::= \lambda x.e : A \to B$
Base Expressions	$ss ::= s \mid i$
Expressions	$e ::= x \mid e : A \mid e_1\ e_2 \mid ss \mid [s]_{C \to D}$
Results	$r ::= e \mid blame$
Value	$v ::= ss : A \mid [s]_A$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$
Frame	$F ::= v\ [] \mid []\ e$
Typing modes	$\Leftrightarrow ::= \Rightarrow \mid \Leftarrow$
Syntactic sugar	$\lambda x.u \equiv \lambda x.e : \star \to \star$

Figure 6 Syntax of the λB^r calculus (syntax that is the same as λB^g in lighter gray).

$\boxed{\Gamma \vdash e \Leftrightarrow A}$ *(New Typing Rules)*

$$\frac{\cdot \vdash s \Rightarrow C \to D \qquad A \to B \nsim C \to D}{\Gamma \vdash [s]_{A \to B} \Rightarrow A \to B}\ \textsc{Etyp-save}$$

Figure 7 Type system of the λB^r calculus. Only new typing rules are shown. All other typing rules are the same as Figure 3.

Contexts, Frame and Typing modes. Typing environments and typing modes are just the same as in the λB^g calculus. Compared to the λB^g calculus, annotation contexts are not in the frame. This change is because in the λB^g calculus we accumulate the annotations, but in the λB^r calculus we employ a blame recovery semantics. If annotated expressions were formulated in the frame, we could not formulate a rule that recovers a saved value.

4.2 Typing

As the λB^g calculus, bidirectional typing is used. Most of the rules are standard and the same as those used by the λB^g calculus in Figure 3. The only novel rule is rule Etyp-save, which states that saved forms s should be well-typed with type $C \to D$, and the type $A \to B$ in the saved expression $[s]_{A \to B}$ is not consistent with type $C \to D$. The context is empty because we only use saved expressions as intermediate results during reduction and such results must be closed.

Dynamic type for the λB^r calculus. As in the λB^g calculus, dynamic types play an important role in the calculus. $\rceil v \lceil$ denotes the dynamic type of v, and $\rceil ss \lceil$ denotes the dynamic type of ss. We need both dynamic types for values and base expressions ss, and we can define dynamic types easily as follows:

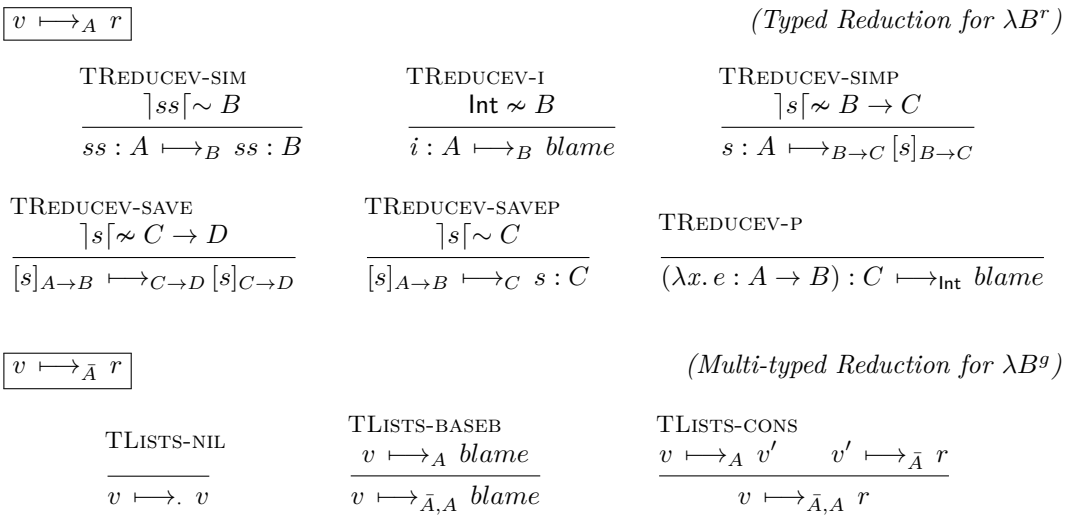


Figure 8 Typed Reduction for the λB^r Calculus.

▶ **Definition 17** (Dynamic type). *$\lceil ss \rceil$ returns the dynamic type of the base expressions ss. $\lceil v \rceil$ returns the dynamic type of the value v.*

$$\begin{aligned} \lceil i \rceil &= \mathsf{Int} \\ \lceil \lambda x.\, e : A \to B \rceil &= A \to B \\ \lceil s : A \rceil &= A \\ \lceil [s]_A \rceil &= A \end{aligned}$$

Two lemmas about dynamic types and a typing lemma about checking mode are:

▶ **Lemma 18** (Dynamic Types of Values). *If $\cdot \vdash v \Rightarrow A$ then $\lceil v \rceil = A$.*

▶ **Lemma 19** (Dynamic Types of Base Expressions). *If $\cdot \vdash ss \Rightarrow A$ then $\lceil ss \rceil = A$.*

▶ **Lemma 20** (Checked expressions can be inferred). *If $\Gamma \vdash e \Leftarrow A$ then $\exists B$, $\Gamma \vdash e \Rightarrow B$.*

4.3 Dynamic Semantics

As in the λB^g calculus, typed reduction is used in the semantics to get a direct operational semantics. Interestingly, the calculus uses not only typed reduction with single type, but also typed reduction for a collection of types.

Typed Reduction. The typed reduction rules are shown in Figure 8. Rule TReducev-sim shows that $ss : A$ type-reduces by type B to $ss : B$, if the type of ss is consistent with type B. If the type of ss is not consistent with type B, the cases for integer (i) and lambda ($\lambda x.\, e : A \to B$), which are included in ss, are different. For $i : A$, rule TReducev-i shows that if type B is not consistent with Int, it raises blame. For a value of the form $(\lambda x.\, e : A_1 \to B_1) : A$, if the type used for typed reduction is a function type $B \to C$, then the value reduces to $[\lambda x.\, e : A_1 \to B_1]_{B\to C}$ using rule TReducev-simp. However

rule TReducev-p says that if the type used for typed reduction is Int, then blame is raised. Rule TReducev-savep says that if the dynamic type of a saved form in $[s]_{A \to B}$ is consistent with C, then we can recover and return $s : C$. Otherwise, if the dynamic type of the saved form s is also not consistent with $C \to D$, then $[s]_{A \to B}$ type-reduces to another saved value $[s]_{C \to D}$ as shown by rule TReducev-save.

An Example. Lets take an example to explain behavior of typed reduction with blame recovery semantics. Suppose that we take a chain of annotations $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \mathsf{Int} \to \mathsf{Int} : \star : \mathit{Bool} \to \mathit{Bool} : \star$. Firstly, the dynamic type of $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}$ is consistent with type $\star$ and the intermediate type $\mathsf{Int} \to \mathsf{Int}$ is erased. Then the dynamic type of $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}$ is not consistent with $\mathit{Bool} \to \mathit{Bool}$. While reducing such an expression, an intermediate saved expression $[\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}]_{\mathsf{Int} \to \mathsf{Int}}$ is generated. However, the saved expression would later be recovered because the final type annotation $\star$ is consistent with the dynamic type of the value.

The typed reduction (and reduction) steps to reduce such an expression are shown next:

$(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \mathsf{Int} \to \mathsf{Int} : \star : \mathit{Bool} \to \mathit{Bool} : \star$
$\longmapsto$ {by STEP-ANNOV and typed reduction under $\star$}
$(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star : \mathit{Bool} \to \mathit{Bool} : \star$
$\longmapsto$ {by STEP-ANNOV and typed reduction under $\mathit{Bool} \to \mathit{Bool}$}
$([\lambda x : \mathsf{Int} \to \mathsf{Int}]_{\mathit{Bool} \to \mathit{Bool}}) : \star$
$\longmapsto$ {by STEP-ANNOV and typed reduction under $\star$}
$(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star$

Typed Reduction Properties. Typed reduction for the λB^r calculus has some interesting properties. The most interesting property is transitivity of typed reduction, which may come as a surprise since the consistency relation is not transitive, and typed reduction for λB^g is not transitive either. The transitivity lemma (Lemma 21) says that typed reduction is the same no matter whether it is type-reduced directly or indirectly via some intermediate type.

▶ **Lemma 21** (Transitivity of typed reduction). *If* $v \longmapsto_A v_1$, *and* $v_1 \longmapsto_B v_2$ *then* $v \longmapsto_B v_2$.

Lets take an example, firstly using the typed reduction of λB^g:

1) $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} \longmapsto_{\star \to \star} \lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} : \star \to \star$
2) $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} : \star \to \star$
$\longmapsto_{\mathit{Bool} \to \mathit{Bool}} \lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} : \star \to \star : \mathit{Bool} \to \mathit{Bool}$
3) $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} \not\longmapsto_{\mathit{Bool} \to \mathit{Bool}} \lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} : \star \to \star : \mathit{Bool} \to \mathit{Bool}$

The three typed reductions correspond to the two premises and the conclusion in the transitivity lemma. The last typed reduction does not hold, and is a counter-example to transitivity of typed reduction in λB^g. Although $\mathsf{Int} \to \mathsf{Int}$ is consistent with $\star \to \star$ and $\star \to \star$ is consistent with $\mathit{Bool} \to \mathit{Bool}$, $\mathsf{Int} \to \mathsf{Int}$ is not consistent with $\mathit{Bool} \to \mathit{Bool}$. Since transitivity does not hold in type consistency and the annotations are accumulated in λB^g, the transitivity of typed reduction does not hold in λB^g. However in λB^r, the annotations are not accumulated and saved expressions are used to save the source type, so the transitivity

of typed reduction holds. The following three typed reductions illustrate what happens for the above example in λB^r:

1) $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} \longmapsto_{\star\to\star} \lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \star \to \star$
2) $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \star \to \star \longmapsto_{Bool\to Bool} [\lambda x.x : Int \to Int]_{Bool\to Bool}$
3) $\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} \longmapsto_{Bool\to Bool} [\lambda x.x : Int \to Int]_{Bool\to Bool}$

Additionally, typed reduction has several of the other properties for typed reduction shown in Section 3:

▶ **Lemma 22** (Preservation of Typed Reduction). *If* $\cdot \vdash v \Leftarrow B$ *and* $v \longmapsto_A v'$ *then* $\cdot \vdash v' \Rightarrow A$.

▶ **Lemma 23** (Progress of Typed Reduction). *If* $\cdot \vdash v \Leftarrow A$ *then* $\exists v'$, $v \longmapsto_A v'$.

▶ **Lemma 24** (Determinism of Typed Reduction). *If* $\cdot \vdash v \Leftarrow B$, $v \longmapsto_A r_1$ *and* $v \longmapsto_A r_2$ *then* $r_1 = r_2$.

Multi-Typed Reduction. The bottom of Figure 8 shows multi-typed reduction. If a value v is multi-type reduced with an empty type collection, then the original v is returned as shown in rule TLists-nil. Rule TLists-baseb states that value v multi-type reducing with a type collection $(\bar{A}, A)$ raises blame when v type-reduced under type A raises blame. Rule TLists-cons says that value v multi-type reducing with a type collection $(\bar{A}, A)$ returns r if v type-reduces under type A return v' and further reduction of v' under $\bar{A}$ returns r.

Reduction. Figure 9 shows the reduction rules of the λB^r calculus. Rule vstep-eval and rule vstep-blame are standard rules. Annotation expressions are not in the frame because we aim at having a blame recovery semantics. Rule vstep-annop and rule vstep-anno are standard rules. Rule vstep-abs and rule vstep-i add an extra annotation with the dynamic type to produce a value. In rule vstep-annov if v type-reduces to v' under type A then $v : A$ reduces to v'.

There are four rules related to beta-reduction. Rule vstep-beta is the main form of beta-reduction. However, the argument v needs to first be (multi)type reduced with the input types, and the annotations with the output types are added in the final expression. If the multi-typed reduction of v raise blame, then the final result is also blame as shown in rule vstep-betap. Rule vstep-apps is another form of beta-reduction that recovers the lambda expression in the saved value when the argument value v successfully type-reduces to another value. Importantly, because the dynamic type of the lambda expression and the saved value are inconsistent, an intermediate type $\star$ is added in between the inputs/output types in both multi-typed reduction and the annotations for the resulting expression. The reason why $\star$ is needed in multi-typed reduction is that without $\star$, the result of typed reduction would not be well-typed. Take an example where v is $1 : \star$ and we are multi-type reducing using the inconsistent types $(Bool, \mathsf{Int})$. The final value $1 : Bool$ is not well-typed:

$1 : \star \longmapsto_{Bool,\mathsf{Int}} 1 : \mathsf{Int} \longmapsto_{Bool} 1 : Bool$ $\qquad\qquad$ $1 : Bool$ is not well-typed!

When the multi-typed reduction of v raises blame, the final result is blame as shown in rule vstep-appsp. Note that an alternative to rule vstep-apps and rule vstep-appsp is to have a rule that always raises blame for any saved expression being applied. Such alternative rule would be significantly simpler, but would raise blame in some cases where

$\boxed{e \longmapsto r}$ *(Small-step Semantics for the λB^r calculus)*

$$\text{VSTEP-EVAL}\ \frac{e \longmapsto e'}{F.\,e \longmapsto F.\,e'} \qquad \text{VSTEP-BLAME}\ \frac{e \longmapsto \mathit{blame}}{F.\,e \longmapsto \mathit{blame}} \qquad \text{VSTEP-ANNOP}\ \frac{e \longmapsto \mathit{blame} \quad \neg(\mathit{value}\ e : A)}{e : A \longmapsto \mathit{blame}} \qquad \text{VSTEP-ANNOV}\ \frac{v \longmapsto_A v'}{v : A \longmapsto v'}$$

$$\text{VSTEP-APPS}\ \frac{v \longmapsto_{A_1,\star,A_2} v'}{([\lambda x.\, e : A_1 \to B_1]_{A_2 \to B_2})\, v \longmapsto e[x \mapsto v'] : B_1 : \star : B_2} \qquad \text{VSTEP-ANNO}\ \frac{e \longmapsto e' \quad \neg(\mathit{value}\ e : A)}{e : A \longmapsto e' : A}$$

$$\text{VSTEP-APPSP}\ \frac{v \longmapsto_{A_1,\star,A_2} \mathit{blame}}{([\lambda x.\, e : A_1 \to B_1]_{A_2 \to B_2})\, v \longmapsto \mathit{blame}} \qquad \text{VSTEP-ABS}\ \frac{}{\lambda x.\, e : A \to B \longmapsto (\lambda x.\, e : A \to B) : A \to B}$$

$$\text{VSTEP-BETA}\ \frac{v \longmapsto_{A_1,A_2} v'}{((\lambda x.\, e : A_1 \to B_1) : A_2 \to B_2)\, v \longmapsto e[x \mapsto v'] : B_1 : B_2} \qquad \text{VSTEP-I}\ \frac{}{i \longmapsto i : \mathsf{Int}}$$

$$\text{VSTEP-BETAP}\ \frac{v \longmapsto_{A_1,A_2} \mathit{blame}}{((\lambda x.\, e : A_1 \to B_1) : A_2 \to B_2)\, v \longmapsto \mathit{blame}}$$

Figure 9 Semantics of the λB^r Calculus.

the blame calculus or λB^g do not. The more complex rules VSTEP-APPS and VSTEP-APPSP are necessary to ensure that blame is not raised when a analogous program in λB^g would not raise blame. The last example in Section 2 shows this situation and illustrates the benefit of having rule VSTEP-APPS to respect the blame semantics of λB^g.

One important property is that the reduction relation is deterministic:

▶ **Theorem 25** (Determinism of λB^r calculus). *If $\cdot \vdash e \Leftarrow A$, $e \longmapsto r_1$ and $e \longmapsto r_2$ then $r_1 = r_2$.*

Type Safety. Another important property is that the λB^r calculus is type safe. Theorem 26 says that if an expression is well-typed with type A, the type will be preserved after the reduction. Progress is shown by Theorem 27. A well-typed expression e will be a value or there exists an expression e' which e could reduce to e' or e could raise blame.

▶ **Theorem 26** (Type Preservation of λB^r Calculus). *If $\cdot \vdash e \Leftrightarrow A$ and $e \longmapsto e'$ then $\cdot \vdash e' \Leftrightarrow A$.*

▶ **Theorem 27** (Progress of λB^r Calculus). *If $\cdot \vdash e \Leftrightarrow A$ then e is a value or $\exists e'$, $e \longmapsto e'$ or $e \longmapsto \mathit{blame}$.*

Less blame. λB^r raises blame strictly less often than λB^g. As we have seen in Section 2 we can find programs that raise blame in λB^g, but will result in values in λB^r. Moreover we have proved the following theorem:

$$
\begin{array}{rcl}
|\,i\,| & = & i \\
|\,\lambda x.\, e : A \to B\,| & = & \lambda x.\, |\,e\,| : A \to B \\
|\,e : A\,| & = & |\,e\,| : A \\
|\,e_1\, e_2\,| & = & |\,e_1\,|\,|\,e_2\,| \\
|\,[s]_{A \to B}\,| & = & |\,s\,| : \star \to \star : A \to B
\end{array}
$$

Figure 10 Translating λB^r expressions to λB^g.

▶ **Theorem 28** (Conformance to the blame semantics of λB^g). *If* $\cdot \vdash |\,e\,| \Leftrightarrow^g A$ *and* $e \longmapsto^r$ *blame then* $|\,e\,| \longmapsto^{g*}$ *blame.*

which states that if a λB^r expression e reduces to blame, and the corresponding λB^g expression $|e|$ is well-typed, then reducing $|e|$ also raises blame. Note that in the theorem, for disambiguation, we annotate the relations with g or r to clarify which calculus does the relation belong to. In other words a program that results in blame in λB^r will also result in blame in λB^g. Moreover, because of the soundness lemma between λB^g and λB, λB^r also raises blame less often than λB.

To prove Theorem 28 we need a translation function between λB^r expressions and λB^g expressions. In λB^r, we have saved expressions/values, while there is no such expression in λB^g. The translation function is shown in Figure 10. For instance, the λB^r expression $[\lambda x.\, x : \mathit{Bool} \to \mathit{Bool}]_{\mathsf{Int} \to \mathsf{Int}}$ would translate to $(\lambda x.\, x : \mathit{Bool} \to \mathit{Bool}) : \star \to \star : \mathsf{Int} \to \mathsf{Int}$ in λB^g.

4.4 Gradual Guarantee

Siek et al. [41] suggested that a calculus for gradual typing should also enjoy the gradual guarantee, which ensures that programs can smoothly move from being more/less dynamically typed into more/less statically typed.

Precision. The top of Figure 11 shows the precision relation on types. $A \sqsubseteq B$ means that A is more precise than B. Every type is more precise than type $\star$. A function type $A_1 \to B_1$ is more precise than $A_2 \to B_2$ if type A_1 is more precise then A_2 and type B_1 is more precise than B_2. The bottom of Figure 11 shows the precision relation of expressions. $e_1 \sqsubseteq e_2$ means that e_1 is more precise than e_2. The precision relation of expressions is derived from the precision relation of types. Every expression has a precision relation with itself. $\lambda x.\, e_1 : A_1 \to B_1$ is more precise than $\lambda x.\, e_2 : A_2 \to B_2$ if e_1 is more precise than e_2 and the types are in the precision relation. For application expressions, precision holds if $e_1 \sqsubseteq e_2$ holds and $e_1' \sqsubseteq e_2'$ holds. For annotated expressions $e_1 : A$ is more precise than $e_2 : B$ if e_1 is more precise than e_2 and A is more precise than B. For saved expressions, the precision relation is similar to annotation expressions.

Notably, a saved expression $[s_1]_{A \to B}$ is more precise than an expression $s_2 : C \to D$ if s_1 is more and precise than s_2 and the type $A \to B$ is more precise than $C \to D$ (rule EP-SA). Lets take an example, to see the use of such precision rule. The expression $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \mathit{Bool} \to \mathit{Bool}$ is more precise than $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \star \to \star$ by rule EP-ANNO. According to the reduction rules, $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \mathit{Bool} \to \mathit{Bool}$ reduces to $[\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}]_{\mathit{Bool} \to \mathit{Bool}}$, while $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \star \to \star$ reduces to $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star$. In addition, $e_1 : \star : A$ is more precise than $e_2 : B$ if e_1 is more precise than e_2 and type A is more precise than B by rule EP-ANNOL. As an example to illustrate the usefulness of this rule the expression $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to$

$\boxed{A \sqsubseteq B}$ *(Precision relation for types)*

$$\text{TP-I}\ \frac{}{\mathsf{Int} \sqsubseteq \mathsf{Int}} \qquad \text{TP-DYN}\ \frac{}{A \sqsubseteq \star} \qquad \text{TP-ABS}\ \frac{A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq B_2}{(A_1 \to B_1) \sqsubseteq (A_2 \to B_2)}$$

$\boxed{e_1 \sqsubseteq e_2}$ *(Precision relation for expressions)*

$$\text{EP-REFL}\ \frac{}{e \sqsubseteq e} \qquad \text{EP-ABS}\ \frac{e_1 \sqsubseteq e_2 \qquad A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq B_2}{\lambda x.\, e_1 : A_1 \to B_1 \sqsubseteq \lambda x.\, e_2 : A_2 \to B_2} \qquad \text{EP-APP}\ \frac{e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{(e_1\, e_2) \sqsubseteq (e_1'\, e_2')}$$

$$\text{EP-ANNO}\ \frac{A \sqsubseteq B \qquad e_1 \sqsubseteq e_2}{e_1 : A \sqsubseteq e_2 : B} \qquad \text{EP-SAVE}\ \frac{A \sqsubseteq C \qquad B \sqsubseteq D \qquad s_1 \sqsubseteq s_2}{[s_1]_{A \to B} \sqsubseteq [s_2]_{C \to D}} \qquad \text{EP-SA}\ \frac{A \sqsubseteq C \qquad B \sqsubseteq D \qquad s_1 \sqsubseteq s_2}{[s_1]_{A \to B} \sqsubseteq s_2 : C \to D}$$

$$\text{EP-ANNOL}\ \frac{A \sqsubseteq B \qquad e_1 \sqsubseteq e_2}{e_1 : \star : A \sqsubseteq e_2 : B} \qquad \text{EP-SAVER}\ \frac{A \sqsubseteq C \qquad B \sqsubseteq D \qquad e_1 \sqsubseteq s_2}{e_1 : \star : A \to B \sqsubseteq [s_2]_{C \to D}}$$

Figure 11 Precision relations.

$\star : \star : \mathit{Bool} \to \mathit{Bool}$ is more precise than $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \mathit{Bool} \to \mathit{Bool}$. The expression $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \star : \mathit{Bool} \to \mathit{Bool}$ reduces to $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star : \mathit{Bool} \to \mathit{Bool}$ while $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \star \to \star : \mathit{Bool} \to \mathit{Bool}$ would reduce to a saved value $[\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}]_{\mathit{Bool} \to \mathit{Bool}}$. Rule EP-SAVER shows the precision relation of these two results. Rule EP-SAVER says that $e_2 : \star : A \to B$ is more precise than $[s_2]_{C \to D}$ while e_1 is more precise than s_2 and type $A \to B$ is more precise than $C \to D$.

Static Gradual Guarantee. Theorem 29 shows that the static criteria of the gradual guarantee holds for the λB^r calculus. It says that if e is more precise than e', e has type A and e' is has type B, then type A is more precise than B.

▶ **Theorem 29** (Static Gradual Guarantee of λB^r Calculus). *If* $e \sqsubseteq e'$, $\cdot \vdash e \Rightarrow A$ *and* $\cdot \vdash e' \Rightarrow B$ *then* $A \sqsubseteq B$.

Dynamic Gradual Guarantee. The λB^r calculus has a dynamic gradual guarantee. Here we formulate a theorem for the dynamic gradual guarantee. Theorem 31 shows that if e_1 is more precise than e_2, e_1 and e_2 are well-typed, and if e_1 reduces to e_1', then e_2 reduces (in multiple steps) to e_2'. Note that e_1' is guaranteed to be more precise than e_2'. Theorem 31 is similar to the one formalized in the AGT approach [17]. A small difference is that we use a multi-step relation in the conclusion because in the precision relation we have rules like rule EP-ANNOL. If we have that $1 : \star : \mathsf{Int}$ is more precise than $1 : \star$, then $1 : \star : \mathsf{Int}$ needs to reduce to $1 : \mathsf{Int}$ while $1 : \star$ is already a value for which no step is required. Theorem 32 is derived easily from Theorem 31. The auxiliary Lemma 30, which shows the property of dynamic gradual guarantee for typed reduction, is helpful to prove Theorem 32.

▶ **Lemma 30** (Dynamic Gradual Guarantee for Typed Reduction). *If* $v_1 \sqsubseteq v_2$, $\cdot \vdash v_1 \Leftarrow A$, $\cdot \vdash v_2 \Leftarrow B$, $A \sqsubseteq B$ *and* $v_1 \longmapsto_A v_1'$ *then* $\exists v_2'$, $v_2 \longmapsto_A v_2'$ *and* $v_1' \sqsubseteq v_2'$.

▶ **Theorem 31** (Dynamic Gradual Guarantee). *If* $e_1 \sqsubseteq e_2$, $\cdot \vdash e_1 \Leftrightarrow A$, $\cdot \vdash e_2 \Leftrightarrow B$ *and* $e_1 \longmapsto e_1'$ *then* $\exists e_2'$, $e_2 \longmapsto^* e_2'$ *and* $e_1' \sqsubseteq e_2'$.

▶ **Theorem 32** (Dynamic Gradual Guarantee). *If* $e_1 \sqsubseteq e_2$, $\cdot \vdash e_1 \Leftrightarrow A$, $\cdot \vdash e_2 \Leftrightarrow B$ *and* $e_1 \longmapsto^* v_1$ *then* $\exists v_2$, $e_2 \longmapsto^* v_2$ *and* $v_1 \sqsubseteq v_2$.

5 Related Work

This section discusses related work. We focus on gradual typing criteria, cast calculi, gradually typed calculi, the AGT approach and typed operational semantics.

Gradual Typing Languages and Criteria. There is a growing number of research work focusing on combining static and dynamic typing [2, 7, 22, 29, 30, 34, 45, 46, 48, 53]. Many mainstream programming languages have some form of integration between static and dynamic typing. These include TypeScript [6], Dart [8], Hack [51], Cecil [10], Bigloo [35], Visual Basic.NET [30], ProfessorJ [20], Lisp [44], Dylan [36] and Typed Racket [50].

Much work in the research literature of gradual typing focuses on the pursuit of sound gradual typing. In sound gradual typing the idea is that some form of type-safety should still be preserved. This often requires some dynamic checks that arise from static type information. Furthermore, gradually typed languages should provide a smooth integration between dynamic and static typing. For instance, one of the criteria for gradual typing is that a program that has static types should behave equivalently to a standard statically typed program [43]. Siek et al. [41], proposed the *gradual guarantee* to clarify the kinds of guarantees expected in gradually typed languages. The principle of the gradual guarantee is that static and dynamic behavior changes by changing type annotations. For the static (gradual) guarantee, the type of a more precise term should be more precise than the type of a less precise term. For the dynamic (gradual) guarantee, any program that runs without errors should continue to do so with less precise types.

Cast calculi. Due to the unknown type and consistency of the gradual typing, more programs are accepted by a gradual type system compared to their analogous static type system. Therefore, some runtime checks are required at run-time to ensure type-safety. The most common approach to give the semantics to a gradually typed language is by translating to a cast calculus, which has a standard dynamic semantics. The process of the translation to cast calculi involves inserting casts whenever type consistency holds.

There are several varieties of cast calculi. Findler and Felleisen [15] introduced assertion-based contracts for higher-order functions. Based on mirrors and contracts, Gray et al. [20] shown a new model to implement Java and Scheme. Henglein's dynamically typed λ-calculus [23] is an extention of the statically typed λ-calculus with a dynamic type and explicit dynamic type coercions. Tobin-Hochstadt and Felleisen [49] presented a framework of interlanguage migration, which ensures type-safety.

Wadler and Findler [52] introduced the blame calculus. The blame comes from Findler and Felleisen's contracts and tracks the locations where cast errors happen using blame labels. Siek et al. [37] explored the design space of higher-order casts. For first-order casts (casts on base types), the semantics is straightforward. But there are issues for higher-order casts (functions): a higher-order cast is not checked immediately. For higher-order casts,

checking is deferred until the function is applied to an argument. After application, the cast is checked against the argument and return value. A cast is used as a wrapper and splitted until the wrapped function is applied to an argument. Wrappers for higher-order casts can lead to unbounded space consumption [24].

There are some different designs for the dynamic semantics for casts calculi in the literature. Herman et al. [24] and Wadler et al. [52] use a lazy error detection strategy. With this strategy, run-time type checking is not performed when a higher-order cast is applied to a value. Instead, lazy error detection coerces the arguments of a function to the target type, and checking is only done when the argument is applied. Siek et al. [43] use a different strategy where checking higher-order casts is performed immediately when the source type is the dynamic type ($\star$). Otherwise, the later strategy is the same as lazy error detection. In the λB^r calculus, we introduce the blame recovery semantics, which is essentially to ignore intermediate type annotations in a chain of type annotations for higher-order functions. The idea is to only raise blame if the initial source type of the value and final target types are not consistent. Otherwise, even if intermediate annotations trigger type conversions, which would not be consistent, the final result can still be a value provided that the initial source and final target types are themselves consistent. This alternative approach has a bounded number of annotations, which avoids the accumulation of type annotations (up-to 2 for higher-order values).

Siek and Wadler [42] introduced threesomes, where a cast consists of three types instead of two types (twosomes) of the blame calculus. The threesome calculus is proved to be equivalent to blame calculus and a coercion-based calculus without blame labels but with space efficiency. The three types in a threesome contain the source, intermediate and target types. The intermediate type is computed by the greatest lower bound of all the intermediate types. For example, in a chain of casts:

$$1 : Int \Rightarrow \star : \star \Rightarrow Int : Int \Rightarrow Int$$

the source type and target are both Int and the intermediate type is computed to be the greatest lower bound of $\star$ and Int, resulting in Int. Compared to our λB^r calculus, function values are twosomes (borrowing Siek and Wadler's terminology). Instead of accumulating annotations, and computing the intermediate types, we simply discard them.

Like λB^r, Castagna and Lanvin [9] propose a calculus that discards annotations for higher-order functions. However their semantics is different. The key difference is that in their semantics intermediate casts are discarded after consistency checks are performed. This means that programs such as (here using our notation):

$$\lambda x.\, x : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int} : \star \to \star : Bool \to Bool : \star \to \star$$

will raise CastErrors (i.e. blame), whereas in λB^g, λB and λB^r that is not the case. Indeed one of the design principles of λB^r is that we do not raise blame when λB^g (and λB) does not (see also Theorem 28). Saved expressions are the key to avoiding raising blame too early (or at all) in λB^r, and are generated when Castagna and Lanvin's calculus would generate blame for higher-order casts. Greenberg [21] introduced similar semantics to Castagna and Lanvin [9]. Blame is also raised in the above example. As λB^r, the intermediate consistent type for a higher-order function will be eliminated in Greenberg [21]. While in Castagna and Lanvin [9]'s work, the consistent intermediate type will be stored.

Finally, various cast calculi have been extended with various of features of practical interest. For instance, Ahmed et al. [3] extended the blame calculus to incorporate polymorphism, based on the dynamic sealing proposed by Matthews et al. [28] and Neis et al. [32].

Gradually Typed Calculi. A gradually typed lambda calculus (GTLC) should support both fully static typed and fully dynamic typed, as well as partially typed ones. Siek and Taha [43] introduced gradual typing with the notion of unknown types $\star$ and type consistency. To support object-oriented languages, Siek and Taha [38] extended the work of Abadi and Cardelli [1] and introduced gradual typing for objects. The semantics of both gradually typed calculi are indirectly defined by typed-directed translation to an intermediate language (a cast calculi). Cast calculi are independent from the GTLC, having their own type systems and operational semantics. The only tie between them is type-directed translation from the source gradually typed language to the cast calculus. In λB^g and λB^r, by using TDOS, the semantics of a GTLC is given directly without translating to any other calculus.

Because runtime checking is needed by a gradually typed language, function types dynamically generate function proxies at runtime in most of gradually typed languages. Therefore the number of proxies in unbound. Herman et al. [24] implemented gradual typing based on coercions and combined adjacent coercions. Thus, space consumption has been limited and the type system was proved to be type-safe. Addressing the space consumption issues of gradual typing has been an ongoing research effort for gradual typing, with many works on the area [16, 24, 37, 42]. The blame recovery semantics circumvents some of the space consumption issues by employing a different semantics.

Abstracting Gradual Typing (AGT). Garcia et al. [17] introduce the abstracting gradual typing (AGT) approach, following an idea by Schwerter [4]. An externally justified cast calculus is not required in AGT. Instead the runtime checks are deduced by the evidence for the consistency judgement. For the static semantics, AGT uses techniques from abstract interpretation to lift terms of the static system to gradual terms. A concretization function is used to lift gradual types to static type sets. After that, a gradual type system can be derived according to the static type system. The gradual type system keeps type safety, and enjoys the criteria of Siek et al. [41]. For the dynamic semantics, the semantics is introduced by reasoning about consistency relations. Gradual typing derivations are represented as intrinsically typed terms [12], which correspond to typing derivations directly.

Similarly to the AGT approach, by using TDOS for the dynamic semantics and a bi-directional type system, we can design a gradually typed language with a direct semantics. While related by the fact that both the AGT approach and TDOS provide means to obtain direct operational semantics for gradually typed languages, the two approaches have different and perhaps complementary goals. The goals of TDOS are more modest than those of AGT, which aims at deriving various definitions for gradually types languages in a systematic manner. In contrast TDOS and our work have no such goals. Our main aim is to adapt the standard and well-known techniques from small-step semantics, into the design of gradually typed languages. We expect that the familiarity and simplicity of the TDOS approach would be a strength, whereas the AGT approach requires some more infrastructure, but the payoff is that many definitions can then be derived. For future work, it would be interesting to see whether it is possible to combine ideas from both approaches. Perhaps having much of the AGT infrastructure, but with an alternative model for the dynamic semantics based on TDOS.

Typed Operational Semantics. In this paper, we use the type-directed operational semantics (TDOS) approach [25]. TDOS was originally used to describe the semantics of languages with intersection types and a merge operator. Like gradual typing, such features require a type-dependent semantics. In TDOS type annotations become operationally relevant and

can affect the result of a program. *Typed reduction* is the distinctive feature in TDOS. Typed reduction is used to provide an operational interpretation to type conversions in the language, similarly to coercions in coercion-based calculi [23]. Our work shows that TDOS enables a direct semantics for gradual typing. In this paper, we explored two possible semantics for gradual typing: one following a semantics similar to the blame calculus, and another with a novel blame recovery semantics. One interesting aspect of the blame recovery semantics is that it avoids some space costs that arise in some cast calculi, while being relatively simple.

There are other variants of operational semantics that make use of type annotations. Types are used in Goguen's typed operational semantics [18] reductions, similarly to TDOS. Typed operational semantics has been applied to various calculi, including simply typed lambda calculi [19], calculi with dependent types [14] and higher-order subtyping [13]. An extensive overview of related work on type-dependent semantics is given by Huang and Oliveira [25].

6 Conclusion

In this work we proposed an alternative approach to give a direct semantics to gradually typed languages without an intermediate cast calculus. Our approach is based on TDOS [25]. TDOS is a variant of small-step semantics where type annotations are operationally relevant and a special relation, called typed reduction, gives an interpretation to such type annotations at runtime. We believe that TDOS can be a valuable technique for language designers of gradually typed languages, giving them a simple and direct way to express the semantics of their language.

We presented two gradually typed lambda calculi: λB^g and λB^r. The λB^g semantics is sound to the semantics of λB. The λB^r calculus explores the large design space in the semantics of gradually typed languages with a new semantics that we call *blame recovery semantics*. This new semantics is more liberal than the semantics of the blame calculus, while still ensuring type-safety and a form of the gradual guarantee.

There is much to be done for future work. Obviously, to prove that TDOS is a worthy alternative to existing cast calculi or other approaches for the semantics of gradually typed languages, many more features should be developed with TDOS. Cast calculi have been shown to support a wide range of features, including blame tracking [52], polymorphism [3], subtyping [38] and various other features [26, 40, 47]. We hope to explore this in the future. Another important line for future work is to see whether the blame recovery semantics provides relevant space efficiency benefits in practice. This would require a well-engineered compiler for gradual typing. Perhaps trying to modify the Grift compiler [27] would be a first step on this direction. Empirical validation and case studies would be necessary.

References

1 Martin Abadi and Luca Cardelli. *A theory of objects*. Springer Science & Business Media, 2012.

2 Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM transactions on programming languages and systems (TOPLAS)*, 13(2):237–268, 1991.

3 Amal Ahmed, Robert Bruce Findler, Jeremy G Siek, and Philip Wadler. Blame for all. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 201–214, 2011.

4 Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 283–295, 2014.

5 Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Java & lambda: a featherweight story. *Logical Methods in Computer Science*, 14(3), 2018.

6 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

7 John Tang Boyland. The problem of structural type tests in a gradual-typed language. *Foundations of Object-Oriented Langauges*, 2014.

8 Gilad Bracha. *The Dart programming language*. Addison-Wesley Professional, 2015.

9 Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–28, 2017.

10 Craig Chambers. The cecil language, specification and rationale, 1993.

11 Avik Chaudhuri. Flow: a static type checker for javascript. *SPLASH-I In Systems, Programming, Languages and Applications: Software for Humanity*, 2015.

12 Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

13 Adriana Compagnoni and Healfdene Goguen. Typed operational semantics for higher-order subtyping. *Information and Computation*, 184(2):242–297, 2003.

14 Yangyue Feng and Zhaohui Luo. Typed operational semantics for dependent record types. *arXiv preprint arXiv:1103.3321*, 2011.

15 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59, 2002.

16 Ronald Garcia. Calculating threesomes, with blame. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 417–428, 2013.

17 Ronald Garcia, Alison M Clark, and Éric Tanter. Abstracting gradual typing. *ACM SIGPLAN Notices*, 51(1):429–442, 2016.

18 Healfdene Goguen. A typed operational semantics for type theory, 1994.

19 Healfdene Goguen. Typed operational semantics. In *International Conference on Typed Lambda Calculi and Applications*, pages 186–200. Springer, 1995.

20 Kathryn E Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. *ACM SIGPLAN Notices*, 40(10):231–245, 2005.

21 Michael Greenberg. Space-efficient manifest contracts. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 181–194, 2015.

22 Lars T Hansen. Evolutionary programming and gradual typing in ecmascript 4 (tutorial). *Lars*, 2007.

23 Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.

24 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167, 2010.

25 Xuejing Huang and Bruno C d S Oliveira. A type-directed operational semantics for a calculus with a merge operator. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

26 Lintaro Ina and Atsushi Igarashi. Gradual typing for generics. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 609–624, 2011.

27 Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G Siek. Efficient gradual typing. *arXiv preprint arXiv:1802.06375*, 2018.

28 Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing. In *European Symposium on Programming (ESOP)*, pages 16–31. Citeseer, 2008.

29 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM SIGPLAN Notices*, 42(1):3–10, 2007.

30 Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*. Citeseer, 2004.

31 Andrew Myers. CS 6110 Lecture 8 Evaluation Contexts , Semantics by Translation, 2013.

32 Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. *ACM Sigplan Notices*, 44(9):135–148, 2009.

33 Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

34 Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. *ACM SIGPLAN Notices*, 47(1):481–494, 2012.

35 Manuel Serrano and Pierre Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *International Static Analysis Symposium*, pages 366–381. Springer, 1995.

36 Andrew Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., 1996.

37 Jeremy Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In *European Symposium on Programming*, pages 17–31. Springer, 2009.

38 Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2007.

39 Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and coercion: together again for the first time. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 425–435, 2015.

40 Jeremy G Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the 2008 symposium on Dynamic languages*, pages 1–12, 2008.

41 Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

42 Jeremy G Siek and Philip Wadler. Threesomes, with and without blame. *ACM Sigplan Notices*, 45(1):365–376, 2010.

43 G Siek Jeremy and Taha Walid. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

44 Guy Steele. *Common LISP: the language*. Elsevier, 1990.

45 T Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: run-time support for reasonable interposition. *ACM SIGPLAN Notices*, 47(10):943–962, 2012.

46 Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in javascript. *ACM SIGPLAN Notices*, 49(1):425–437, 2014.

47 Asumu Takikawa, T Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 793–810, 2012.

48 Satish Thatte. Quasi-static typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 367–381, 1989.

49 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 964–974, 2006.

50 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. *ACM SIGPLAN Notices*, 43(1):395–406, 2008.

51 Julien Verlaguet. Facebook: Analyzing php statically. *Commercial Users of Functional Programming (CUFP)*, 13, 2013.

52 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, pages 1–16. Springer, 2009.

53 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *European Conference on Object-Oriented Programming*, pages 459–483. Springer, 2011.

54 Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

Linear Promises: Towards Safer Concurrent Programming

Ohad Rau
Georgia Institute of Technology, Atlanta, GA, USA

Caleb Voss
Georgia Institute of Technology, Atlanta, GA, USA

Vivek Sarkar
Georgia Institute of Technology, Atlanta, GA, USA

Abstract

In this paper, we introduce a new type system based on linear typing, and show how it can be incorporated in a concurrent programming language to track ownership of promises. By tracking write operations on each promise, the language is able to guarantee exactly one write operation is ever performed on any given promise. This language thus precludes a number of common bugs found in promise-based programs, such as failing to write to a promise and writing to the same promise multiple times. We also present an implementation of the language, complete with an efficient type checking algorithm and high-level programming constructs. This language serves as a safer platform for writing high-level concurrent code.

2012 ACM Subject Classification Software and its engineering → Concurrent programming languages; Theory of computation → Operational semantics; Theory of computation → Type theory

Keywords and phrases promises, type systems, linear typing, operational semantics, concurrency

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.13

Supplementary Material *Software (Source Code)*: `https://github.com/OhadRau/LinearPromises` archived at `swh:1:dir:311764ac58400c3720161e108bb3611fcab4c2d9`
Software (ECOOP 2021 Artifact Evaluation approved artifact): `https://doi.org/10.4230/DARTS.7.2.15`

1 Introduction

In recent decades, the prevalence of concurrent programming[1] has increased as programmers strive to take advantage of increasingly parallel machines. Unstructured concurrency has proven to be highly error-prone, and programmers have subsequently undertaken efforts to formalize concurrent programming using structured techniques. One such technique is the *promise* – a container used to refer to a value that is produced asynchronously [14]. Promises typically include both a "read" (or "await") operation – which blocks until a value is available and returns that value – as well as a "write" (or "fulfill") operation – which provides the value for the promise. Promises are frequently used to communicate between threads or tasks, where one thread awaits a value and another performs some asynchronous computation before writing the value to the promise. This creates a higher-level abstraction as compared to more traditional concurrency primitives like the lock/mutex (which allows programs to ensure mutually-exclusive access to shared resources) and a more general abstraction than the future (which represents a placeholder for the result of a function being evaluated by a predetermined task or thread). In recent years, promises have been incorporated into mainstream languages such as C++, JavaScript, and Java as popular concurrency primitives.

[1] In this paper, we use "concurrent" interchangeably with "parallel", though we recognize that a distinction is made between the two terms in other contexts.

Despite the value of structured concurrency in preventing bugs, these techniques – including the promise – are not without their own share of bugs. Madsen et al. [15] identify several common promise-related concurrency bugs, including the *omitted write* – a promise that is never fulfilled – and the *double write* – a promise that is fulfilled multiple times. In their case study, these two bugs are the direct cause of 6 of the 21 concurrency bugs analyzed.

In addition to the omitted write and double write bugs, there are other kinds of concurrency errors that can occur with the use of promises. A *deadlock* is composed of a cyclical dependency among a set of threads [11]. Further, an *unowned write* occurs when one thread assumes ownership over a promise only to have another thread perform the actual write operation. This is distinct from the double write, as it can make it difficult for the programmer to reason about the source of a promise's fulfillment and often results in a race to *fulfill* the promise between two threads. This comes in contrast to a double write to an owned promise, which would only exhibit a race when reading[2].

Each of these bugs has the potential to create critical issues in a program, ranging from making the program inoperable to creating major security flaws. For example, both omitted writes and deadlocks can cause a program to hang indefinitely, while unowned or double writes may contribute to data races and, by extension, inconsistent, unpredictable, or even undefined behavior. In fact, these bugs have been cited [9, 10] as a disadvantage of promises relative to futures, which by definition cannot result in unowned, double, or omitted writes[3].

Over the years, programmers have devised many tools that can be used to quickly detect bugs in programs. In general, these fall into two major categories: dynamic analyses, which attempt to identify bugs during a program's execution, and static analyses, which attempt to identify bugs before execution [8]. Each method has its own advantages, with dynamic analyses generally producing fewer false positives as well as more detailed results (due to the availability of more information about the program) and static analyses revealing possible bugs before the code is run [7]. Beyond that, static analyses present the advantage of proving the *absence* of behaviors (i.e. that a program can *never* exhibit a certain bug). This allows for bugs to be found that would only appear intermittently or rarely in the running program; in contrast, such bugs can be very challenging to find using dynamic analyses.

One particularly useful class of static analysis takes the form of type checking. *Substructural* type systems are a class of type systems in which restrictions are placed on the number of times a variable may be used. One such system is known as *linear* typing, in which linear variables must be used *exactly once* [23]. As an example, with a linear variable x both the program $x + x$ and 0 cannot be typed, as neither uses the variable x exactly once. In recent years, substructural type systems have risen in popularity through their use in languages such as Rust [24], Haskell [2], ATS [26], and F* [21].

The contributions of this paper are as follows:

1. We present the design and implementation of a new concurrent programming language aimed at preventing certain classes of promise-related bugs.
2. We introduce a linear type system that tracks ownership of promises.
3. We provide a formal semantics for the behavior of concurrent, promise-based programs.
4. We prove that it is *impossible* to create double writes and unowned writes in this language, as well as omitted writes in terminating programs.

[2] The write permission of an owned promise is only owned by one thread at a time, so no two writers can race to fulfill the same *owned* promise. Both situations can result in a race when *reading*, because the value read depends on the non-deterministic order of read and write operations.

[3] Recall that a future can only be fulfilled as the result of a function. Given a function f known to return exactly once, a future yielding f's result will be fulfilled exactly once (and only by f).

5. We provide a decidable type checking algorithm, and experimentally evaluate its speed.
6. We demonstrate the abilities of this language to find bugs in real-world programs.

2 Language Formalisms

Our work revolves around a custom programming language intended to create safer type-level primitives for interacting with promises. However, we believe that this approach can be incorporated into other strongly-typed, concurrent languages with promises. Voss and Sarkar [22] introduce the concept of promise ownership as a key mechanism for dynamically identifying omitted write bugs and deadlocks. Specifically, ownership of a promise p by a task T is defined as the responsibility of T to either fulfill p or transfer ownership to another task. Based on their approach, we leverage a linear type system to statically track promise ownership. By using the type system, we can not only *statically* prevent omitted writes in terminating programs, but also unowned and double writes. As in their work, we can also leverage the language's semantics to *dynamically* detect deadlocks.

This notion of promise ownership is established by splitting promises into two components, a "read-end" and a "write-end," similar to the split between "futures" and "promises" in C++, respectively [12]. In this system, the write-end of a promise is linearly typed so as to ensure it is fulfilled exactly once. Compound data types, such as products and sums, are linear if they contain one or more linear components. All other variables remain unrestricted. By applying linear typing in this way, we enable liberal use of the read-end, while restricting the write-end to only a single use by its owner. We formally define the programming language in three parts (syntax, type system, and semantics), taking careful consideration to create as simple a language as possible.

2.1 Syntax

The language consists of multi-procedure programs, which are able to spawn asynchronous tasks. Values are integers, promise read and write handles, sums, products, and named functions. Control flow is expressed by matching on sums and products, while looping is expressed via recursion. Functions take a single parameter and return a single value. Types take the form of the base Int type for integers, the promise read handle type $\mathsf{Promise}(\tau)$, the promise write handle type $\mathsf{Promise}^*(\tau)$, binary sum and product types[4], and function types.

Figure 1 defines the syntax, and features several interesting constructs:

- $\mathsf{async}\ e$ – Schedules the expression e to be performed asynchronously.
- $\mathsf{promise}\ \tau$ – Constructs a promise of a τ, yielding a pair of the write and read handles.
- $?x$ – Blocks on and retrieves the value contained by a promise x (e.g. "await").
- $x \leftarrow e$ – Writes the value of e to the owned promise x (resolving/fulfilling the promise).

2.2 Type System

Before specifying the typing rules, we must first introduce the notion of a linear type [23]. Variables of linear types have the property that they must be used exactly once along any path of execution. For example, if a linear variable x is in scope, then the expression 1 cannot be typed, since it does not use the linear variable x. Likewise, the expression $x + x$

[4] Unbounded data was left out of the language definition for simplicity. The language can be trivially extended with recursive data types (Section 4.1) to allow for dynamically-sized data types. Such a modification does not invalidate any of the language's properties.

$$\tau \in \mathsf{Type} ::= \mathsf{Int} \mid \mathsf{Promise}(\tau) \mid \mathsf{Promise}^*(\tau) \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2$$

$$\begin{aligned} e \in \mathsf{Expr} ::= {} & i \mid x \mid \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2 \mid f(e) \mid \mathsf{async}\ e \mid (e_1, e_2) \\ & \mid \mathsf{Inl}_{\tau_L,\tau_R}\ e \mid \mathsf{Inr}_{\tau_L,\tau_R}\ e \\ & \mid \mathsf{match}\ e_0\ \{\ (x_1, x_2) \Rightarrow e_1\ \} \\ & \mid \mathsf{match}\ e_0\ \{\ \mathsf{Inl}\ x_1 \Rightarrow e_1\ ,\ \mathsf{Inr}\ x_2 \Rightarrow e_2\ \} \\ & \mid \mathsf{promise}\ \tau \mid ?x \mid x \leftarrow \mathsf{e} \end{aligned}$$

$$F \in \mathsf{Function} ::= \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \{\ e\ \}$$

$$P \in \mathsf{Program} ::= F \mid F\ P$$

Figure 1 Syntax rules for the language.

$$\frac{}{\textsc{IsLinear}(\mathsf{Promise}^*(\tau))}$$

$$\frac{\textsc{IsLinear}(\tau_1) \vee \textsc{IsLinear}(\tau_2)}{\textsc{IsLinear}(\tau_1 \times \tau_2)} \qquad \frac{\textsc{IsLinear}(\tau_1) \vee \textsc{IsLinear}(\tau_2)}{\textsc{IsLinear}(\tau_1 + \tau_2)}$$

Figure 2 The IsLinear judgment, which is used to determine whether a type τ is linear.

cannot be typed, since it uses the linear variable x twice. The linear types in our language are types of the form $\mathsf{Promise}^*(\tau)$ (i.e. write handles) and any compound type containing at least one $\mathsf{Promise}^*$. Note that functions in this type system need not be linear, as the lack of closures and global variables ensures that a call to a function cannot use a linear variable without receiving it as a parameter or creating it. A generalization to allow closures is possible, and would require functions to become linear when capturing linear variables[5]. Types are determined to be linear by the judgment IsLinear, defined in Figure 2.

We define the typing environment Γ as a sequence of statements of the form $x : \tau$, such that each statement denotes the type τ of the variable x in the environment. Note that the same environment contains both linearly and non-linearly typed variables.

$$\Gamma \in \mathsf{Environment} ::= \emptyset \mid \Gamma, x : \tau$$

Following the notation of Slepak [20], Figure 3 defines the "environment splitting" relation $\boxplus$, such that given an environment Γ, one can state that $\Gamma = \Gamma_1 \boxplus \Gamma_2$ if the environment Γ can be split into Γ_1 and Γ_2. This splitting relation asserts that the *linear* variables in the environments Γ_1 and Γ_2 are *disjoint* and that each linear variable in Γ belongs to *exactly one* of Γ_1 and Γ_2. However, *non-linear* variables *can* be duplicated between both Γ_1 and Γ_2. For example, given a linear variable v and a standard variable w, we cannot split the environment in such a way as to share the variable v between two scopes (i.e. if $\Gamma = \Gamma_1 \boxplus \Gamma_2$, then $v \in \Gamma_1 \not\veebar v \in \Gamma_2$). However, the same restriction does not apply to w, which can be present in one or both environments.

[5] A linear closure would be restricted to a single call. This is overly conservative in some cases, but has been successfully employed in other languages with substructural type systems (e.g. FnOnce in Rust).

$$\frac{}{\emptyset = \emptyset \boxplus \emptyset} \qquad \frac{\Gamma = \Gamma_1 \boxplus \Gamma_2}{\Gamma, x : \tau = (\Gamma_1, x : \tau) \boxplus \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \boxplus \Gamma_2}{\Gamma, x : \tau = \Gamma_1 \boxplus (\Gamma_2, x : \tau)}$$

$$\frac{\Gamma = \Gamma_1 \boxplus \Gamma_2 \qquad \neg \textsc{IsLinear}(\tau)}{\Gamma, x : \tau = (\Gamma_1, x : \tau) \boxplus (\Gamma_2, x : \tau)}$$

Figure 3 The environment splitting relation $\boxplus$.

$$\text{(T-Weaken)}\ \frac{\Gamma_1 \vdash e : \tau_1 \qquad \nexists x : \tau_2 \in \Gamma_2 . \textsc{IsLinear}(\tau_2)}{\Gamma_1 \boxplus \Gamma_2 \vdash e : \tau_1}$$

$$\text{(T-Int)}\ \frac{}{\vdash i : \mathsf{Int}} \qquad \text{(T-Var)}\ \frac{}{x : \tau \vdash x : \tau} \qquad \text{(T-Let)}\ \frac{\Gamma_1 \vdash e_v : \tau_v \qquad \Gamma_2, x_v : \tau_v \vdash e_b : \tau_b}{\Gamma_1 \boxplus \Gamma_2 \vdash \mathsf{let}\ x_v := e_v\ \mathsf{in}\ e_b : \tau_b}$$

$$\text{(T-App)}\ \frac{f : \tau \to \tau' \vdash f : \tau \to \tau' \qquad \Gamma \vdash e : \tau}{\Gamma \boxplus f : \tau \to \tau' \vdash f(e) : \tau'} \qquad \text{(T-Async)}\ \frac{\Gamma \vdash e : \mathsf{Int}}{\Gamma \vdash \mathsf{async}\ e : \mathsf{Int}}$$

$$\text{(T-Product)}\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1 \boxplus \Gamma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \text{(T-Left)}\ \frac{\Gamma \vdash e_L : \tau_L}{\Gamma \vdash \mathsf{Inl}_{\tau_L, \tau_R}\ e_L : \tau_L + \tau_R}$$

$$\text{(T-Right)}\ \frac{\Gamma \vdash e_R : \tau_R}{\Gamma \vdash \mathsf{Inr}_{\tau_L, \tau_R}\ e_R : \tau_L + \tau_R}$$

$$\text{(T-MatchProduct)}\ \frac{\Gamma_1 \vdash e_v : \tau_1 \times \tau_2 \qquad \Gamma_2, x_1 : \tau_1, x_2 : \tau_2 \vdash e_b : \tau_b}{\Gamma_1 \boxplus \Gamma_2 \vdash \mathsf{match}\ e_v\ \{\ (x_1, x_2) \Rightarrow e_b\ \} : \tau_b}$$

$$\text{(T-MatchSum)}\ \frac{\Gamma_1 \vdash e_v : \tau_L + \tau_R \qquad \Gamma_2, x_L : \tau_L \vdash e_L : \tau_b \qquad \Gamma_2, x_R : \tau_R \vdash e_R : \tau_b}{\Gamma_1 \boxplus \Gamma_2 \vdash \mathsf{match}\ e_v\ \{\ \mathsf{Inl}\ x_L \Rightarrow e_L\ ,\ \mathsf{Inr}\ x_R \Rightarrow e_R\ \} : \tau_b}$$

$$\text{(T-Promise)}\ \frac{\neg \textsc{IsLinear}(\tau)}{\vdash \mathsf{promise}\ \tau : \mathsf{Promise}^*(\tau) \times \mathsf{Promise}(\tau)} \qquad \text{(T-Read)}\ \frac{\Gamma \vdash x : \mathsf{Promise}(\tau)}{\Gamma \vdash ?x : \tau}$$

$$\text{(T-Write)}\ \frac{\Gamma_1 \vdash x_p : \mathsf{Promise}^*(\tau) \qquad \Gamma_2 \vdash e_v : \tau}{\Gamma_1 \boxplus \Gamma_2 \vdash x_p \leftarrow e_v : \mathsf{Int}}$$

$$\text{(T-Func)}\ \frac{x : \tau \vdash e : \tau'}{\vdash \mathsf{fun}\ f(x : \tau) : \tau' \{e\} : \tau \to \tau'}$$

Figure 4 The typing rules for the language[a].

[a] The results of (T-Async) and (T-Write) are meaningless, but evaluate to placeholder integers due to the absence of a void/unit type.

Listing 1 Program violating double write restriction.

```
1 fun doubleWrite(p: Promise*(Int)): Int {
2   let _ := p ← 0 in
3   p ← 1
4 }
```

Figure 4 defines the typing rules. While most of the rules are fairly standard, it is important to observe the mechanics used for linear typing. Each rule only allows the most restricted possible environment[6] to be present (e.g. (T-Var) operates on an environment that contains only the referenced variable). This prevents the program from dropping a linear variable without using it.

The (T-Weaken) rule is employed to relax the restriction of only applying a rule to the smallest environment. Specifically, (T-Weaken) allows for unrestricted variables to be dropped arbitrarily. For example, take the program $\mathsf{let}\ x := 5\ \mathsf{in}\ 0$. Since the (T-Int) rule only operates on an empty environment, we cannot directly apply it to type 0 as an Int with x in the context. However, (T-Weaken) allows us to drop the variable x when type-checking 0, because x is an unrestricted variable. In practice, this means that unrestricted variables behave exactly as they would in languages without linear typing. Note that (T-Weaken) *cannot* be applied to linear variables, since dropping a linear variable would allow it to escape from being used (breaking the guarantee that each linear variable is used exactly once).

Environment splitting accomplishes the other half of linear typing. Whenever a linear variable appears in the environment for an expression containing multiple sub-expressions, we must ensure that it is only available in one sub-expression. For example, consider a product (e_1, e_2) and a linear variable x. If x appeared in both e_1 and e_2, it would clearly allow for x to be used twice. To circumvent this, typing rules with multiple sub-expressions *split* the environment into several sub-environments, such that each linear variable appears in only one sub-environment. In this case, x could only be in scope for one of e_1 and e_2.

Several rules relate directly to promises and are critical to implementing the desired safety guarantees. Specifically, note the fact that the promise constructor in (T-Promise) returns both a linear/owned handle for the promise and a standard handle for the promise; this creates the split between the write-end and read-end of the promise, respectively. In the case of (T-Read), observe that it cannot accept a write-end to a promise, as reading from the write-end would "use" the promise and result in never writing to it. Likewise, observe that in (T-Write) the promise must be a linear write-end, so as to enforce only writing to a promise a single time and only in an owned context.

Put together, these typing rules enforce the critical property that promises must be written to exactly once. Since only one un-copyable and un-droppable handle exists for writing to a given promise, it is impossible to write to a promise twice or to forget to write to a promise. Further, the establishment of ownership by the linear type system enforces the property that a promise can only ever be written to by its owner.

As an illustration of the typing rules and their effects, let us consider several small programs that fail to type check and, more importantly, demonstrate bugs that can occur via incorrect use of promises.

[6] This is similar to the "Small Footprint" rule used in separation logics, in that the typing rules contain the smallest possible environment but can be generalized by (T-Weaken).

Listing 2 Program violating omitted write restriction.

```
1 fun omittedWrite(p: Promise*(Int)): Int {
2   match Inl_{Int,Int} 0 {
3     Inl i ⇒ 0,
4     Inr i ⇒ p ← 1
5   }
6 }
```

Listing 3 Program violating unowned write restriction.

```
1 fun unownedWrite(p: Promise(Int)): Int {
2   p ← 0
3 }
```

Listing 1 illustrates a double write bug. When type checking this function, we must split the environment on line 2 (via (T-Let)) such that p belongs to the right-hand side of the let expression and is no longer owned in line 3. Thus, it becomes impossible to type check line 3 in this context, since (T-Write) demands that p be owned when writing. In other words, there is *no* way to type this function. If allowed to execute, this program would attempt to assign a second value to the promise p. This may lead to undefined behavior; if, for example, two threads await the promise p, they may read different values depending on when the second write occurs. While other languages handle this scenario as a runtime error – for example, this same error is handled as an exception future_errc::promise_already_satisfied in C++ and by simply ignoring subsequent writes to the promise in JavaScript – our language can statically prevent this behavior from *ever* occurring.

Listing 2 demonstrates an omitted write bug, in which a promise is not fulfilled in some paths of execution. As an analogy, compare this to how the Java compiler rejects functions that are missing a return statement in one branch of a conditional. When type checking the match expression on line 2 with (T-MatchSum), we split the environment between the value being matched and the two branches, requiring that both branches type check under the same environment. In this case, we assign an empty environment to the value being matched and an environment consisting of p to the two branches. While the branch on line 4 type checks via (T-Write), the variable p is unused on line 3. Since p is a linear variable, it is impossible to eliminate p via (T-Weaken), and thus the program cannot type check. When executing this program, the match expression would reach line 3 and fail to assign a value to p before returning.

Finally, Listing 3 demonstrates writing to a promise that is not owned. This is in contrast to Listing 1, where a write is attempted while *no* promise is in scope. When attempting to type check line 2, we find it impossible to apply (T-Write) since the type of p is not of the form Promise*(Int). During execution, this program would allow various bugs to occur due to the inability to track this write operation elsewhere in the program. For example, it could perform a second write to an already-fulfilled promise, creating a double write bug.

2.3 Operational Semantics

To define the operational semantics of a program, we must first introduce some extra syntax to represent the runtime state of a program. First, we extend the expression syntax with the construct $\&i_\tau$, which represents a reference to the read-end of a promise of τ with the

$$\text{(T-PromiseRef)}\ \frac{}{\vdash \&i_\tau : \mathsf{Promise}(\tau)} \qquad \text{(T-PromiseRef}^*\text{)}\ \frac{}{\vdash \&^* i_\tau : \mathsf{Promise}^*(\tau)}$$

Figure 5 Typing rules for promise references.

$$m \in \mathsf{Thread\ ID} \quad ::= \ i$$
$$T \in \mathsf{Threads} \quad ::= \ \langle m, e \rangle \ \mid \ T \parallel T$$

Figure 6 Syntax for thread trees.

unique ID i. We also add the construct $\&^* i_\tau$, which represents a reference to the write-end of a promise of τ with the unique ID i. We augment the type system with two new rules, defined in Figure 5, to give the promise-reference literals a type.

Figure 6 defines the currently-executing state of a program as a binary tree of threads, known as a *fork-tree*. In such a fork-tree, each node contains a thread ID and an expression. Each fork operation splits the node to a left thread, representing the currently executing code, and a right thread, representing the new task. These terms are composed together to create a tree of all concurrently executing tasks. We encode the entire state of the program as the quadruple (i, P, O, t): the next-available ID for a promise or thread, a mapping of promise IDs to an optional value, a mapping of promises to their owner thread, and the thread tree for the program. Semantically, P stores the values (or lack thereof) associated with a given promise, and also indicates whether a given promise has been fulfilled. This is encoded by the type $\tau + \mathsf{Int}$, where τ corresponds to a value and Int corresponds to a placeholder for an unfulfilled promise (the value of which is ignored). The initial state of a program is $(1, \emptyset, \emptyset, \langle 0, \mathit{main}(0) \rangle)$, where $\mathit{main} : \mathsf{Int} \to \mathsf{Int}$.

Figure 7 defines the IsValue predicate, used to identify expressions that are fully evaluated. Finally, we utilize contextual semantics to simplify the notation for the operational semantics. Specifically, Figure 8 defines two contexts K and J to represent an expression with a hole and a fork-tree with a hole, respectively. While the normal transition rules (e.g. (S-Write)) implement the transition rules for base-case expressions, the K contextual semantics allow us to recursively apply these transformations to sub-expressions and the J contextual semantics allow us to apply these transformations to any thread in the tree.

Figure 9 defines the operational semantics. Of particular interest here are the rules for modeling non-determinism. The binary fork-tree structure of the program allows us to apply the basic stepping rules (such as function application, let evaluation, etc.) to any thread at

$$\frac{}{\text{IsValue}(i)} \qquad \frac{}{\text{IsValue}(\&i_\tau)} \qquad \frac{}{\text{IsValue}(\&^* i_\tau)} \qquad \frac{\text{IsValue}(v_1) \wedge \text{IsValue}(v_1)}{\text{IsValue}((v_1, v_2))}$$

$$\frac{\text{IsValue}(v)}{\text{IsValue}(\mathsf{Inl}_{\tau_L, \tau_R}\ v)} \qquad \frac{\text{IsValue}(v)}{\text{IsValue}(\mathsf{Inr}_{\tau_L, \tau_R}\ v)}$$

Figure 7 The IsValue predicate, which determines whether an expression is a terminal value.

$$
\begin{aligned}
K \in \text{Expr Context} \quad ::= \; & \cdot \mid \mathsf{let}\ x := K\ \mathsf{in}\ e \mid f(K) \mid \mathsf{Inl}\ K \mid \mathsf{Inr}\ K \\
& \mid (K, e) \mid (e, K) \mid \mathsf{match}\ K\ \{\ \mathsf{Inl}\ x_1 \Rightarrow e_1, \mathsf{Inr}\ x_2 \Rightarrow e_2\ \} \\
& \mid \mathsf{match}\ K\ \{\ (x_1, x_2) \Rightarrow e\ \} \mid ?K \mid K \leftarrow e \mid e \leftarrow K \\
J \in \text{Thread Context} ::= \; & \cdot \mid J \parallel T \mid T \parallel J
\end{aligned}
$$

Figure 8 K (expression-level) and J (thread-level) contextual semantics.

$$
\text{(SJ-Schedule)}\ \frac{(i, P, O, \langle m, e\rangle) \longrightarrow (i', P', O', T')}{(i, P, O, J[\langle m, e\rangle]) \longrightarrow (i', P', O', J[T'])}
$$

$$
\text{(SK-RunExpr)}\ \frac{(i, P, O, \langle m, e\rangle) \longrightarrow (i', P', O', \langle m, e'\rangle)}{(i, P, O, \langle m, K[e]\rangle) \longrightarrow (i', P', O', \langle m, K[e']\rangle)}
$$

$$
\text{(SK-RunFork)}\ \frac{(i, P, O, \langle m, e\rangle) \longrightarrow (i', P', O', \langle m, e'\rangle \parallel T)}{(i, P, O, \langle m, K[e]\rangle) \longrightarrow (i', P', O', \langle m, K[e']\rangle \parallel T)}
$$

$$
\text{(S-Fork)}\ \frac{\Gamma \vdash e : \tau \qquad O_L = \{p \mapsto m \mid p \mapsto m \in O, p \notin \Gamma\} \qquad O_R = \{p \mapsto i \mid p \mapsto m \in O, p \in \Gamma\}}{(i, P, O, \langle m, \mathsf{async}\ e\rangle) \longrightarrow (i+1, P, O_L \cup O_R, \langle m, 0\rangle \parallel \langle i, e\rangle)}
$$

$$
\text{(S-Let)}\ \frac{\textsc{IsValue}(v)}{(i, P, O, \langle m, \mathsf{let}\ x := v\ \mathsf{in}\ e\rangle) \longrightarrow (i, P, O, \langle m, e[v/x]\rangle)}
$$

$$
\text{(S-App)}\ \frac{\textsc{IsValue}(v)}{(i, P, O, \langle m, f(v)\rangle) \longrightarrow (i, P, O, \langle m, \textsc{Body}(f)[v/\textsc{Arg}(f)]\rangle)}
$$

$$
\text{(S-MatchProduct)}\ \frac{\textsc{IsValue}(v_1) \qquad \textsc{IsValue}(v_2)}{\begin{array}{c}(i, P, O, \langle m, \mathsf{match}\ (v_1, v_2)\{(x_1, x_2) \Rightarrow e\}\rangle) \longrightarrow \\ (i, P, O, \langle m, e[v_1/x_1, v_2/x_2]\rangle)\end{array}}
$$

$$
\text{(S-MatchSumL)}\ \frac{\textsc{IsValue}(v)}{\begin{array}{c}(i, P, O, \langle m, \mathsf{match}\ \mathsf{Inl}\ v\{\mathsf{Inl}\ x_L \Rightarrow e_L, \mathsf{Inr}\ x_R \Rightarrow e_R\}\rangle) \longrightarrow \\ (i, P, O, \langle m, e_L[v/x_L]\rangle)\end{array}}
$$

$$
\text{(S-MatchSumR)}\ \frac{\textsc{IsValue}(v)}{\begin{array}{c}(i, P, O, \langle m, \mathsf{match}\ \mathsf{Inr}\ v\{\mathsf{Inl}\ x_L \Rightarrow e_L, \mathsf{Inr}\ x_R \Rightarrow e_R\}\rangle) \longrightarrow \\ (i, P, O, \langle m, e_R[v/x_R]\rangle)\end{array}}
$$

$$
\text{(S-Promise)}\ \frac{}{(i, P, O, \langle m, \mathsf{promise}\ \tau\rangle) \longrightarrow (i+1, [i \mapsto \mathsf{Inr}\ 0; P], [i \mapsto m; O], \langle m, (\&^* i_\tau, \&i_\tau)\rangle)}
$$

$$
\text{(S-Read)}\ \frac{e_p = \&(i_p)_\tau \qquad P(i_p) = \mathsf{Inl}\ v}{(i, P, O, \langle m, ?e_p\rangle) \longrightarrow (i, P, O, \langle m, v\rangle)}
$$

$$
\text{(S-Write)}\ \frac{e_p = \&^*(i_p)_\tau \qquad \textsc{IsValue}(v)}{(i, P, O, \langle m, e_p \leftarrow v\rangle) \longrightarrow (i, [i_p \mapsto \mathsf{Inl}\ v; P], O, \langle m, 0\rangle)}
$$

Figure 9 Operational semantics for the language.

any time, while also enabling forking to occur at any program point. In other words, we rely on the ability to *substitute* a single thread (e.g. $\langle \mathsf{let}\ x := \mathsf{async}\ e_1\ \mathsf{in}\ e_2 \rangle$) with a sub-tree (e.g. $\langle \mathsf{let}\ x := 0\ \mathsf{in}\ e_2 \rangle \parallel \langle e_1 \rangle$) to allow for forking within the scope of an arbitrary expression.

Much of the heavy-lifting with regards to concurrency occurs via the contextual semantics, which are implemented via three transition rules: (SJ-Schedule), (SK-RunExpr), and (SK-RunFork). (SJ-Schedule) forms the basis of the threading model, allowing us to treat any individual task in the fork-tree as a hole (where we can apply other transition rules). In practice, this implements scheduling for the program by picking a thread and performing transformations on it. The two K-level rules, (SK-RunExpr) and (SK-RunFork) are extremely similar, in that they focus on a hole in the expression tree and apply further transformations to it. In practice, these holes take the form of larger expressions that require reduction before evaluation; for example, consider how (S-App) requires the function's argument to be reduced to a value before calling the function. The two rules differ in that (SK-RunFork) specifically allows for a fork operation to occur within the sub-expression, whereas (SK-RunExpr) runs the sub-expression as a singular thread. As an example, imagine that the argument to a function involves an async operation. When evaluating the argument, the hole is no longer a single expression, but rather a fork-tree of two expressions. When substituting this subtree into the program, care must be taken to ensure that the old thread's expression is substituted in while the new thread stays distinct.

The rule (S-Fork) is also of particular interest, as it is used to spawn new threads. When (S-Fork) creates a new thread, it must assign a new thread ID to it (picking a unique value from the state of the program i). Note that a promise write handle's owner is the thread that will write to that promise. That is, since two thread IDs now exist, each promise in scope will be mapped to one of these IDs in the ownership map O. For example, if the newly spawned thread will use an owned promise p then we must transfer the ownership of p to the new thread. This runtime ownership tracking is used only for proofs, and can be removed from the semantics of the language for an actual implementation without any effect. At the same time, it provides a tool that could be leveraged to perform deadlock detection at runtime [22], which may be useful for an implementation.

Finally, let us briefly consider the mechanics of the (S-Promise), (S-Read) and (S-Write) rules. These rules all rely on the map P, which tracks each promise's state. (S-Promise) adds an entry for i (the ID of the new promise), mapping it to $\mathsf{Inr}\ 0$ – a placeholder indicating that p hasn't been fulfilled. In (S-Write), P is extended with a mapping of i_p (the ID of the promise p) to $\mathsf{Inl}\ v$ (a value v). The use of the Inl constructor indicates that p has been fulfilled, and the value v corresponds to p's new value. (S-Read) checks the state of i_p, and only advances once the value stored in $P(i_p)$ is fulfilled (indicated by the Inl constructor). Once p is fulfilled, (S-Read) evaluates to the stored value. In effect, this forces (S-Read) to block until a value for p becomes available.

3 Theoretical Guarantees

The goal of our language is designing a system that is free of a number of common concurrency bugs. Specifically, we argue that our language cannot exhibit an omitted write bug (creating a promise but never fulfilling it) in terminating programs, and cannot ever exhibit an unowned write bug (writing to a promise that is not owned by the current thread) or a double write bug (fulfilling the same promise twice).

▶ **Theorem 1** (Linearity). *Given a linear variable $x : \tau_x$, if $\Gamma, x : \tau_x \vdash e : \tau_e$ then it is always the case that $\Gamma \vdash e : \tau_e$ cannot be well-typed. Likewise, if $\Gamma \vdash e : \tau_e$ then $\Gamma, x : \tau_x \vdash e : \tau_e$ cannot be well-typed. That is, a linear variable cannot be added/dropped in Γ for the same e.*

Proof. We argue that whenever a linear variable $x : \tau_x$ exists in the environment $\Gamma, x : \tau_x$ for a well-typed program e, then x cannot be dropped while retaining a well-typed program. Further, whenever $x : \tau_x$ *does not exist* in Γ for a well-typed e, x cannot be added to Γ while retaining a well-typed program. Each can be shown by trivial induction on the typing rules. ◀

3.1 Soundness

We define soundness to mean that if a program type-checks it can only get "stuck" due to promise dependency cycles. That is, either a program is done executing, is able to progress via the operational semantics, or it contains a set of threads T such that every thread in T is blocked on a promise owned by another thread in T. This is critical to proving the various other properties of our language.

▶ **Lemma 2** (Substitution Preservation). *Given a well-typed program, such that* $\Gamma \vdash e : \tau$*, if* $\Gamma \vdash x_0 : \tau_0$ *and* $\Gamma \vdash e_0 : \tau_0$ *then* $\Gamma \vdash e[e_0/x_0] : \tau$*.*

Proof. Observe that the typing relation is defined such that if an expression is well-typed, any sub-expression must also be well-typed. Since x_0 and e_0 share the same type, substitution cannot effect the overall typing judgment. This can be trivially shown by induction on e. ◀

▶ **Lemma 3** (Promise Preservation). *Given a promise* $p : \mathit{Promise}(\tau)$*,* $\vdash P(p) : \tau + \mathit{Int}$*.*

Proof. First, observe that an owned promise $p : \mathsf{Promise}^*(\tau)$ can only be fulfilled via the (T-Write) rule on an expression $p \leftarrow v$, where the value v is an inhabitant of the type τ. By definition, the value *written* to the promise will always match the correct type τ.

Next, observe that the write and read ends of a promise always share their type. The only way to create a promise literal of the form $\&i_\tau$ or $\&^*i_\tau$ is via the (S-Promise) rule. By definition this rule assigns the same *unique* ID to both ends, so each newly created read-end and write-end will correspond to a matching type τ.

Finally, notice that the type of both a read-end and a write-end of a promise will never change throughout the program. Via (T-PromiseRef) and (T-PromiseRef*), a promise literal's type is directly encoded in the syntax of the promise literal. By case analysis, it is trivially true that there is no way to transform the type encoded in the promise literal. There is no possible syntax that can allow for the wrong type of value to be written to a promise and there is also no possible syntax to attempt reading a promise as the wrong type.

Thus, in any step $e \longrightarrow e'$ where e involves a write operation, both the value placed into the promise and the promise itself always retain their types. Therefore it can be seen that no possible sequence of steps can result in a promise p containing a value of the wrong type. ◀

▶ **Lemma 4** (Local Preservation). *Given a program* $\langle m, e\rangle$ *such that* $\vdash e : \tau$*, then if* $\langle m, e\rangle \longrightarrow \langle m, e'\rangle$ *or* $\langle m, e\rangle \longrightarrow \langle m, e'\rangle \parallel t$ *it must be true that* $\vdash e' : \tau$*.*

Proof. By simple induction over the transition rules and appeal to Lemmas 2 and 3, it can be seen that any step $e \longrightarrow e'$ yields e' with the same type τ (and when forking, a second thread with any type). Therefore, in all cases the original thread retains its type τ. ◀

▶ **Theorem 5** (Global Preservation). *For any well-typed program, any operational semantics step taken will leave us with a well-typed program where each thread either retains its old type or has been newly created.*

Proof. By induction on the step that is taken, observe that either:

1. The program consists of a single thread and does not fork, in which case it is trivially true that the program's type remains globally preserved via Lemma 4.
2. A new thread is created and the parent thread is modified via (S-Fork). By Lemma 4 we know that the parent thread is preserved. Since the newly created thread was not previously present in the tree, and the sub-expression it is created from was well-typed, then it must be true that the new thread is well-typed.
3. A thread is modified in a program consisting of many threads, which can only be done by using the thread-level contextual semantics to navigate to that thread's subtree. In this case we can apply the inductive hypothesis to the thread's subtree in order to show that all threads in the subtree are preserved once they are modified. The threads that were not modified are preserved by definition, therefore it must be true that the entire tree is preserved after substituting in the modified subtree. ◀

▶ **Lemma 6** (No Ownership on Termination). *For any thread $\langle m, e\rangle$ with well-typed e, e cannot be reduced to a terminal value while still owning a promise.*

Proof. Observe that, by definition, all terminal values can only be well-typed in the empty environment. We cannot drop an owned promise (or any linear variable that might contain a promise) via (T-Weaken). No task can terminate as a linear value, as the async expression requires a type of Int for the spawned task and the *main* function evaluates to Int (i.e. all threads evaluate to Int). Thus, before termination a thread must *always* eliminate all owned promises. ◀

▶ **Definition 7** (ContainsCycle(O, T)). *Given a promise-ownership map O and a tree of tasks T, let G be a graph containing a vertex for each task in T. Then, for every task t_1 that is currently reading a promise p, where p's owner $O(p) = t_2$, create a directed edge from t_1 to t_2. The pair (O, T) are said to contain a cycle iff G contains a cycle.*

▶ **Theorem 8** (Progress). *Given a well-typed program (i, P, O, T), either $(i, P, O, T) \longrightarrow (i', P', O', T') \vee \forall t \in T.\textsc{IsValue}(t) \vee \textsc{ContainsCycle}(O, T)$.*

Proof. We structure our progress theorem as the statement that given a well-typed program either the program can make progress, the program has terminated (i.e. every thread is a value literal), or the program contains a promise dependency cycle. In any case where a step can be made or all threads have terminated, there is nothing to show. Thus, if we assume that progress is impossible but the program has not terminated, we argue that it must be due to the existence of such a promise dependency cycle.

Let us assume that all non-value threads in T are not steppable and that at least one thread $t = \langle m, e\rangle$ is not reduced to a value. For a non-value thread to be un-steppable, it must be performing a read operation applied to a promise reference: $K[?(\&i_\tau)]$. This can be seen by induction over e: if e is not a terminal value, the program can always unconditionally step forward or reduce further using K-level contextual semantics *unless* it is in the form of a read operation $K[?(\&i_\tau)]$ where the promise $\&i_\tau$ is unfulfilled (in which case e is *blocked* on the promise i). Let the owner thread $o = O(i)$, the owner of the promise i. We consider a recursive walk through the tree of threads, in which there are three cases:

1. o refers to the current thread. In this case, we trivially observe a cycle from o to o.
2. o refers to a thread that is a basic value and cannot be further evaluated. This case is trivially shown to be impossible, since o is unable to own the promise i via Lemma 6.
3. o refers to another blocked thread, and we recursively apply the same logic to o.

Given that there is a finite set of threads, by the pigeonhole principle we observe that the process of traversing dependencies must eventually terminate with a cycle (note that we have already rejected the base case of the thread being able to step forward). By this logic, it can be seen that the only case where a program is not terminated and cannot step forward is when a cycle exists in (O, T).

Therefore we have shown that for any well-typed program, the condition holds that the program is finished evaluating, can step forward, or contains a dependency cycle. ◀

▶ **Theorem 9** (Soundness). *Given a well-typed program* (i, P, O, T)*, if* $(i, P, O, T) \longrightarrow^* (i', P', O', T')$ *then either the program* T' *can continue, all threads have reached a terminal value, or it contains a cycle.*

Proof. By induction over the set of steps $(i, P, O, T) \longrightarrow^* (i', P', O', T')$:

1. If no steps were taken, we argue that since the program is well-typed then by Theorem 8 it must be true that either T' can step forward, it has terminated, or it contains a cycle.
2. If at least one step $(i, P, O, T) \longrightarrow (i', P', O', T')$ has been taken, then we apply Theorem 5 to show that the program (i', P', O', T') remains well-typed. Now, by the inductive hypothesis we argue that the rest of the sequence of steps must either be able to continue, have already terminated, or contain a cycle.

Therefore, for any sequence of steps it must be the case that the final result can continue, has terminated, or contains a cycle. ◀

3.2 Additional Properties

Building on Theorem 9, we can easily show that each of the language's guarantees still holds.

▶ **Lemma 10** (Single-Write of Promises). *For any well-typed program, all promises in that program are written to* at most *once. For any well-typed program that* terminates successfully*, all promises in that program are written to* exactly *once.*

Proof. This can be easily observed, as there are only two ways that an owned promise can be used: transferring ownership via a function call/function return/alias or writing to it. Since the single-use property of linear variables is unconditionally true in the program by Theorem 1, then it must be the case that, for any promise p:

1. Given a write operation, a promise p is used and can no longer be reused.
2. Given a transfer operation, a promise p is used and an alias p' is created, which must then be used by the program.
3. Given any other operation, the promise p remains available and must be used.

In order for the above to reach termination, it must eventually reach the base case of writing to a promise. Therefore, for any program that has successfully terminated, all promises have been written to exactly once. Until the point of termination, there is no operation that allows for the duplication of a promise, and thus we can create an upper-bound of at most one write to any promise for programs that have not yet terminated. ◀

▶ **Corollary 11** (No Omitted Writes). *If a program is well-typed and terminates successfully, all created promises will be fulfilled during execution.*

▶ **Corollary 12** (No Double Writes). *If a program is well-typed, no promise will be fulfilled multiple times.*

▶ **Theorem 13** (No Unowned Writes). *If a program is well-typed, any promise that is written to will be owned by the currently executing thread (i.e. the writer).*

Proof. Promises can only become owned by the currently executing thread when created in it or when transferred to it. Transfer can only occur via async, as the only other inter-thread communication consists of promises (which cannot contain linear values). By definition (S-Fork) transfers ownership to the thread that will use the promise. Since the only promise we can write to is an owned promise, and other values cannot change types to become an owned promise by Theorem 5, then it is impossible to write to anything except a promise owned by the current thread. ◀

We have shown that our language does not permit double or unowned writes, and that it does not permit omitted writes in terminating programs. Note that a diverging program may indefinitely postpone promise writes; for example, consider a program that runs an infinite loop before issuing a write to a promise. While this is a limit to the language's guarantees, observe that futures have this same limitation: a future created from a function that does not terminate will never be fulfilled. In effect, the type system gives promises similar safety characteristics to futures: both promises and futures will always be fulfilled at most once and only by their owner, and will always be fulfilled exactly once if the program reaches termination. This side-steps the safety gap between promises and futures noted by [9, 10].

▶ **Corollary 14** (Only Cyclical Deadlocks). *If a program is well-typed, the program can only ever get stuck*[7] *due to a cyclical deadlock.*

Proof. This directly follows from Theorem 9. ◀

With the result of Corollary 14, it's useful to note the advantages our language presents for runtime deadlock detection. Voss and Sarkar [22] present an algorithm for dynamically detecting deadlocks in promise-based programs, which consists of two conditions: that all promises are fulfilled and that there are no cyclical dependencies among tasks caused by promises awaiting a result. Since all promises are known to be eventually fulfilled by Corollary 11, simply identifying the cycles at runtime must be sufficient to catch all possible deadlocks encountered during a program's execution. Notably, the thread/promise-ownership table used for deadlock detection in their algorithm is already tracked at runtime per the operational semantics of the language. In effect, this allows us to trivially implement dynamic deadlock detection in our language without altering the syntax, type system, or semantics.

4 Implementation

In order to evaluate our language, we implemented a compiler that performed type checking and translated our language to a subset of Java[8]. Specifically, the compiler built upon the formalisms of the language and carefully extended it to allow for more user-friendly programming. Thanks to the type system's guarantees, no additional runtime overhead is added to the generated programs.

[7] In the sense that no operational semantics steps can be taken and the program is not done executing.
[8] The implementation is included as part of the supplementary materials.

4.1 Language Extensions

In designing the compiler, a number of additional features were added to the language that are not represented in Section 2. First, we extended the language to allow named and *recursive* types, as the current language definition only supports anonymous, non-recursive sum and product types. Through a simple syntax for defining new types (known as "records" and "unions"), the language is able to apply similar typing rules to the specified product and sum types, respectively. The addition of named types follows the existing type system rules, so that $\mathsf{IsLinear}(\tau)$ is true for all user-defined τ that contain linear members. In other words, we can still reason about a named type being linear, since naming a type does not hide its linearity. This generalization enables users of the language to create and interact with various recursive types, such as linked-lists and binary trees.

As another quality of life improvement, the language implementation was generalized to allow N-ary functions, whereas the type system described in Section 2 only allows for unary functions (functions with a single parameter). This, combined with support for additional types such as `Unit`, `Bool` and `String`, eases translation for many real-world programs.

Both `for` and `while` loops were also added to the language. However, due to the nature of loops – which may execute any number of times – it is impossible to guarantee that variables in a loop will be used exactly once. For example, consider the program `while condition() { p <- 0 }`. If p is a value of type $\mathsf{Promise}^*(\mathsf{Int})$, we can observe two possible bugs in the above program. Firstly, the function `condition()` may initially return `false`, in which case the write would *never* occur. Likewise, it is possible that `condition()` returns `true` multiple times in a row, in which case the write would occur *more than once*. Both of these possibilities would clearly violate the language's guarantees, and thus must be disallowed. To do so, the compiler prevents capturing a free linear variable in a loop. In this case, unless p is defined within the loop, it is impossible to reference it from within the loop. Note that while such a restriction is overly conservative (i.e. a loop *could* be designed to only write once), similar restrictions exist in other substructural type systems. In practice, such a restriction has not prevented the adoption of other substructurally-typed languages (e.g. Rust, which has partially addressed this problem through the use of iterators).

The compiler also provides a function unsafeWrite, which enables writing to a promise's read-end. At runtime, this operation is equivalent to writing to the write-end, but allows for multiple writes to the same promise during type checking. This enables an escape hatch, which can be useful when directly translating a foreign program or dealing with conditions that are difficult to reason about (e.g. consider a loop that only fulfills a promise on the first iteration). While this construct introduces an unowned write bug (and potentially double writes), it *does not* negate the language's soundness and still prevents the omitted write bug. This must be the case because of the existence of the write-end, which ensures that at least one write unconditionally occurs. If unsafeWrite were implemented as a no-op, the program would still be sound by definition (it is equivalent to the program without unsafeWrite). Now observe that there is no way for an additional write to cause the program to get stuck, because threads can *only* block while awaiting an *unfulfilled* promise. Therefore, it must be the case that all well-typed programs remain sound with unsafeWrite enabled.

4.2 The Type Checking Algorithm

The type checking algorithm is largely built upon the notion of environment splitting. However, to perform splitting efficiently a $\textsc{split}(\Gamma, e_1, e_2)$ function had to be devised such that $\textsc{split}(\Gamma, e_1, e_2) = (\Gamma_1, \Gamma_2) \implies \Gamma = \Gamma_1 \boxplus \Gamma_2 \wedge \Gamma_1 \vdash e_1 : \tau_1 \wedge \Gamma_2 \vdash e_2 : \tau_2$. In

other words, the SPLIT function would inspect two expressions and assign variables from the environment to each expression, creating their corresponding type checking environments. This is used to to mimic the behavior of the $\Gamma_1 \boxplus \Gamma_2$ relation in the type system.

In Voss and Sarkar [22], user-supplied annotations indicate the transfer of promise ownership between tasks. These annotations take the form of a list of promises to "move" at every async expression, and the newly-spawned task becomes the owner of the provided promises. This mechanism is integral to both the dynamic detection of omitted writes and the dynamic deadlock detection algorithm that Voss and Sarkar introduce. By encoding ownership information in the type system and *inferring* the "moves" via our environment splitting algorithm, our type checker provides a way to lower the burden of explicit annotations for such a system. The key, in this case, is the ability of the split operation to statically determine which sub-expression (the async task or the code that follows spawning the new task) should become the owner of any given promise and, through the linearity constraints of promises, force this ownership model to be adhered to.

Algorithm 1 Environment Splitting between Expressions.

1: **function** SPLIT(Γ, e_1, e_2)
2: $\quad \text{free}_1 \leftarrow \mathsf{free}(e_1)$
3: $\quad \text{free}_2 \leftarrow \mathsf{free}(e_2)$
4: $\quad \Gamma_1 \leftarrow \{v : \tau \mid v : \tau \in \Gamma, v \in \text{free}_1\}$
5: $\quad \Gamma_2 \leftarrow \{v : \tau \mid v : \tau \in \Gamma, v \in \text{free}_2\}$
6: $\quad \text{allUsed} \leftarrow \forall v : \tau \in \Gamma \;.\; v \in \Gamma_1 \vee v \in \Gamma_2 \vee \neg \mathsf{IsLinear}(\tau)$
7: $\quad \text{noneReused} \leftarrow \nexists v : \tau \in \Gamma \;.\; \mathsf{IsLinear}(\tau) \wedge v \in \Gamma_1 \wedge v \in \Gamma_2$
8: $\quad$ **if** allUsed $\wedge$ noneReused **then return** (Γ_1, Γ_2)
9: $\quad$ **else type error**

The SPLIT function, defined in Algorithm 1, operates on an environment Γ and two expressions, e_1 and e_2. It begins by finding the free variables in each expression with respect to an empty environment, revealing which variables will be referenced in each expression (lines 2-3). Then, it creates an environment for each expression by taking all variables from Γ that are in that expression's free set (lines 4-5). We then define allUsed as the statement that every variable in Γ either appears in one of the two environments or is non-linear; that is, *are 0 linear variables dropped?* Next, we define noneReused to be the statement that there are no variables in Γ that are linear and used in both environments; that is, *are 0 linear variables copied?* If both conditions hold true, we return the two environments (line 8); otherwise, we throw a type error, as the environment cannot be validly split (line 9).

Since functions are generalized to be N-ary, special care must be taken to ensure that the environment is split correctly for each argument. Algorithm 2 thus defines a special form of the SPLIT function called SPLIT_SEQUENCE(Γ, es). This function utilizes the same approach as the SPLIT function, but operates recursively on a single expression at a time. In doing so, it is able to confirm that at every step the all-used and none-reused conditions still hold.

SPLIT_SEQUENCE functions as a generalization of SPLIT, taking an environment Γ and a list of expressions es. This algorithm recursively walks through es and splits the environment at each expression. The base case of an empty list returns an empty list of environments (line 2). Otherwise, the function continues to its recursive step on line 4. First, we define expr to be the head of the list and rest to be the remainder of the list (lines 4-5). We then find the free variables for expr (line 6) and define free_rest to be the union of the free variables for each expression in rest (line 7). As with SPLIT, we construct a Γ for each set of free variables

Algorithm 2 Environment Splitting between a Sequence of Expressions.

```
1:  function SPLIT_SEQUENCE(Γ, es)
2:      if es = [] then return []
3:      else
4:          expr ← head(es)
5:          rest ← tail(es)
6:          free_expr ← free(expr)
7:          free_rest ← ⋃_{e∈rest} free(e)
8:          Γ1 ← {v : τ | v : τ ∈ Γ, v ∈ free_expr}
9:          Γ2 ← {v : τ | v : τ ∈ Γ, v ∈ free_rest}
10:         allUsed ← ∀v : τ ∈ Γ . v ∈ Γ1 ∨ v ∈ Γ2 ∨ ¬IsLinear(τ)
11:         noneReused ← ∄v : τ ∈ Γ . IsLinear(τ) ∧ v ∈ Γ1 ∧ v ∈ Γ2
12:         if allUsed ∧ noneReused then return append(Γ1, SPLIT_SEQUENCE(Γ2, rest))
13:         else type error
```

(lines 8-9) and then check whether all linear variables are used and none of them are reused (lines 10-12). If so, the current expression can be split, so we return Γ_1 appended to the result of SPLIT_SEQUENCE on rest with Γ_2 (line 12). Otherwise, we throw a type error to signal that the environment cannot be split (line 13).

At a high-level, the type checker performs pattern matching against the various syntactic constructs. Each case is handled as per the typing judgments, utilizing the SPLIT and SPLIT_SEQUENCE functions to assign the environments for each sub-expression. For the base cases of the type checker, special care must be taken to ensure that weakening is applied correctly; in this case, splitting the environment between the actual expression and an empty/dummy expression ensures that only used variables (including linear variables) remain. In the case of branches, both branches must be split from the condition/matched value (and not from each other) so that it is ensured that the same linear variables occur in each branch.

In Algorithm 3, the TYPECHECK function takes an environment Γ and an expression e to type check in Γ. The function defines a variable complete as the result of splitting Γ between e and a dummy expression (line 2). This is used to determine whether all linear variables in Γ are used by e: if e cannot be split, it must be due to dropping or reusing a linear variable in Γ. Next, the function performs pattern matching against e (line 3) to determine the syntactic construct that appears.

Lines 4-7 handle references to variables (called x, in this case). Because this is a base case, care must be taken to ensure that no linear variables are dropped; this is done by checking that complete references a valid environment (line 5). If so, the type of x is fetched from the environment Γ (line 6); otherwise, the function throws a type error (line 7).

Lines 8-12 handle the let syntax. In particular, when a let expression is encountered, the environment must be split so that the value and body are checked in different environments. Γ is split into Γ_{rhs} and Γ_{body}, which are used for the RHS and the body, respectively (line 9). We recurse to determine the type of the RHS (line 10). We then add $id : \tau_{rhs}$ to the body's environment so that the bound variable id is available in the body's scope (line 11). Finally, we return the body's type by recursively calling TYPECHECK (line 12).

Lines 13-22 handle the if syntax. When an if statement is encountered, the environment splitting is a little more difficult. For example, consider the case where one branch uses a linear variable and the other does not; clearly, this would violate the language's guarantees. Thus, we must ensure that both branches use all linear variables that are not assigned to the

Algorithm 3 Type Checker Implementation for Selected Constructs.

```
1:  function TYPECHECK(Γ, e)
2:      complete ← SPLIT(Γ, e, ())
3:      match e with
4:          case x ⇒
5:              if complete is not a type error then
6:                  return Γ(x)
7:              else type error
8:          case let id := rhs in body ⇒
9:              Γ_rhs, Γ_body ← SPLIT(Γ, rhs, body)
10:             τ_rhs ← TYPECHECK(Γ_rhs, rhs)
11:             Γ'_body ← Γ_body, id : τ_rhs
12:             return TYPECHECK(Γ'_body, body)
13:         case if cond then thenBranch else elseBranch ⇒
14:             Γ_cond,0, Γ_then ← SPLIT(Γ, cond, thenBranch)
15:             Γ_cond,1, Γ_else ← SPLIT(Γ, cond, elseBranch)
16:             if Γ_cond,0 = Γ_cond,1 ∧ TYPECHECK(Γ_cond,0, cond) = 'Bool' then
17:                 τ_then ← TYPECHECK(Γ_then, thenBranch)
18:                 τ_else ← TYPECHECK(Γ_then, elseBranch)  ▷ Must both typecheck in Γ_then
19:                 if τ_then = τ_else then
20:                     return τ_then
21:                 else type error
22:             else type error
23:         case f(args...) ⇒
24:             match Γ(f) with
25:                 case (τ_expected ...) → τ' ⇒
26:                     Γ_args ← SPLIT_SEQUENCE(Γ, args)
27:                     τ_args ← {TYPECHECK(Γ_args,i, args_i) | i ∈ 0..LENGTH(args)}
28:                     if τ_args = τ_expected then
29:                         return τ'
30:                     else type error
31:                 otherwise ⇒ type error
32:             end match
33:         case async e ⇒
34:             if TYPECHECK(Γ, e) = 'Unit' then
35:                 return 'Unit'
36:             else type error
37:         ...   ▷ Other cases have been omitted for brevity, but follow a similar approach
38:     end match
```

condition. We begin by splitting the environment between *cond* and *thenBranch*, and *cond* and *elseBranch*, respectively (lines 14-15). This leaves us with two valid environments for *cond* and guarantees that the two branches are split in such a way that they cannot drop any linear variables. We then check whether the two *cond* environments are identical and that *cond* evaluates to a boolean (line 16). If not, we throw a type error (line 22). Next, we check that the two branches evaluate to the same type (line 19); if so, we return that type (line 20) and if not, we throw a type error (line 21).

Lines 23-32 handle function calls. Note that in contrast to the typing rules presented earlier, functions here are N-ary. As a result, *args* is a list of N arguments to the function f. We begin by looking up the type of f in Γ to confirm it is indeed a function, calling its argument types $\tau_{expected}$ and return type τ' (lines 24-25). Whenever f cannot be found in Γ or it is not a function type, we throw a type error (line 31). To type check the arguments, we need each to have its own environment. To do so, we call the SPLIT_SEQUENCE function on *args* to split Γ for each argument (line 26). Then, we map over each argument and recursively call TYPECHECK on it with the corresponding environment, storing the results in τ_{args} (line 27). When τ_{args} matches $\tau_{expected}$, we return the function's return type τ' (lines 28-29); otherwise, we throw a type error (30).

Lines 33-36 handle the async syntax. Because async only contains a single sub-expression, knowing that the sub-expression is well-typed is enough to ensure that the entire async expression is also well-typed; thus we do not need to check the complete property. Given an expression e to run asynchronously, we simply TYPECHECK it in Γ and verify that it evaluates to the unit type (line 34). If so, we can return the unit type for the entire async expression (line 35). Otherwise, we signal a type error (line 36).

The various other cases are left out for brevity, as they use the same techniques demonstrated in the above cases.

Since the TYPECHECK function visits each syntax node exactly once, and at each syntax node performs a SPLIT operation, the time complexity is a linear function of the cost of type checking and the cost of splitting. The complexity of splitting is proportional to the size of the environment (i.e. the number of variables in scope) and the time complexity of the free function. With memoization, it is possible to create an amortized $O(1)$ implementation of the free function since the same nodes are repeatedly visited (the worst-case – an unvisited node – being linear with respect to the size of the subtree). Thus, we believe that with an efficient implementation of free the amortized worst-case time complexity of TYPECHECK is $O(|\Gamma| \times |e|)$, where Γ is the environment and e is the syntax tree.

5 Evaluation

We evaluate our compiler implementation on two separate metrics. First, we measured the speed at which a number of test programs of various sizes could be type checked. This was important to evaluate *how practical* opting into such a type system would be. Second, we conduct a case study on the language's ability to catch bugs. This is done by translating a number of JavaScript programs containing promise-related bugs. In performing this case study, we are able to evaluate *how useful* opting into such a type system would be.

5.1 Type Checker Performance

While a theoretical worst-case time complexity was calculated for the type checker, we were also interested in its the real-world performance. To estimate the scalability of the type checking algorithm in realistic conditions, we created several synthetic test programs that

Table 1 Compile time evaluation.

Program	Lines of Code	Type Checking μs per run[1)]	Full Compile μs per run[1)]	% time in type checker[2)]
Infer	5	6.5	9.1	71.4%
Strings	6	26.3	28.8	91.3%
State	12	25.3	28.6	88.5%
Square	26	58.9	68.6	85.9%
Useful	29	59.5	64.8	91.8%
Basic	34	57.1	67.2	85.0%
Cppreference	38	76.9	109.6	70.2%
IO	45	189.5	211.6	89.6%
Long	231	1107.0	1359.8	81.4%

1) Averaged over 1000 runs.
2) The compiler performs a straight-shot translation to source-level Java, and thus requires very little time to generate code. Type-checking therefore comprises the bulk of the work.

demonstrated various language constructs (available in the supplementary materials). These programs ranged in length from 5 lines to 231 lines, and ranged in complexity from simple tests to programs that implemented buffered, asynchronous reading and parsing of files. To measure the performance, we created a benchmarking mode in the compiler. This mode allowed us to measure runtime for 1000 runs of both the type checker and the entire compiler pipeline. Care was taken to time *only* the operation in question, so that for example parsing time would not contribute to the type checking benchmark and the overhead of file I/O syscalls would never contribute to the full pipeline benchmark. Thus, all necessary input was pre-computed to allow for a simple loop of the runs to be timed together.

Table 1 shows the average runtime for type checking and compiling each program, as well as the proportion of the total compile time spent type checking. The benchmarks were run in Windows 10 on an Intel i5-7300U CPU clocked to 2.71GHz with 8GB of RAM.

In general, we find that the results in Table 1 are empirically consistent with an approximately linear average time complexity with respect to the length of the program. However, variance does exist due to the actual factors (number of syntax nodes and variables in scope) not necessarily scaling proportionally to the overall length of programs. For example, consider the worst-case for our type checker: a program composed of one or more extremely long functions, which introduce hundreds of variables into scope. In such a program, we might expect approximately quadratic time with respect to the length of the program due to the cost of splitting the environment at each syntax node. Overall, we believe that our type checker is sufficiently performant for most real-world use cases, as our benchmarks show that the performance typically exceeds the worst-case time complexity.

5.2 Case Study

Madsen et al. [15] perform a case study of common promise-based bugs in JavaScript using programs from StackOverflow. To evaluate our compiler, we attempted to port these programs to our language; while not all features were possible, we tried to capture the general behavior of each program. For example, whenever callbacks were registered as event listeners, we replaced these with asynchronous infinite loops that would conditionally call a function representing the callback. A number of programs could not be translated to our language, as they relied on various JavaScript features that could not be emulated. Several programs used

Table 2 Case study of JavaScript promise bugs posted to StackOverflow.

StackOverflow ID	Type of Bug	Detected?
41268953	Omitted Write	✓
41488363	Omitted Write	✓
42304958	Double Write	✓
42551854	Double Write	✓
42672914	Omitted Write	✓
42777771	Double Write	✓
29210234	Fork in Promise Chain	
41699046	Missing Return in `.then`	✓
41764399	`.then` replaces error	
42163367	Fork in Promise Chain	
42408234	Missing Return leads to Omitted Write	✓
42577647	Failure to Return Promise	✓
42719050	Missing Return leads to Omitted Write	✓
42788603	Missing Return leads to Omitted Write	✓
42828856	Missing Return leads to Omitted Write	✓

exceptions, which our language does not support. Several other programs passed incorrect arguments to the `.then()` method, making their semantics nonsensical (e.g. using a promise value when a function was expected). The source code of the successfully translated programs is included in the `stackoverflow/` directory of the supplementary materials.

Madsen et al. [15] identify 6 out of 21 programs as either omitted writes or double writes, though 4 other programs also exhibit omitted writes indirectly. The additional omitted writes were all caused by a missing return value in the `.then()` method; the callback for this method is expected to return a new value for the promise, which was missing in several programs. Table 2 shows the complete set of tested questions (sourced from the original case study [15]). Since we have proven that these bugs cannot exist in our language (Corollaries 11 and 12), we expected that these programs would not type check.

Listing 4 Translation of StackOverflow question 42777771 with corresponding error.

```
1 func doIt(): Promise(String) begin
2   let numKeys = 1 in
3   promise p, resolve: String in
4   resolve <- "resolve called!";
5   for key = 0 to numKeys begin
6     resolve <- "inside resolve called"
7   end;
8   p
9 end
```

```
Cannot reuse linear variables [resolve: Promise*(String)] in both
  resolve <- "resolve called!"
and
  for key = 0 to numKeys begin
    resolve <- "inside resolve called"
  end;
  p
```

All 10 programs exhibiting omitted or double writes failed to compile due to a linearity error under our type checker (e.g. Listing 4). Two other programs failed through basic type errors, due to type mismatches between the branches of if statements.

6 Related Work

6.1 Program Analysis

In the past few decades, numerous approaches have been devised for program analysis [8]. Empirical studies have demonstrated the widespread use of program analysis tools in industrial software engineering [6]. Program analysis techniques can be divided into two classes: dynamic analysis, which occurs during program execution, and static analysis, which occurs without executing the program. While dynamic analysis techniques are generally able to make more precise conclusions due to additional information available only at runtime, their results cannot be generalized to program paths that are not executed in testing. In contrast, static analysis often provides more conservative (and sound) results, usually with fewer or no false negatives [7]. Christakis and Bird [6] observe that "60% [of survey respondents] reported that they would accept a slower analyzer if it captured more issues (fewer false negatives)" and "50.7% felt that finding more real issues was worth the cost of dealing with false positives", showing that programmers may often find value in the more conservative nature of static analysis. In the same study, approximately one third of respondents stated that they would like program analyzers to detect concurrency bugs.

A large body of research has been conducted into dynamic analysis for concurrency bugs [22, 18, 1, 17]. While these works offer promising results in precisely finding concurrency-related bugs, dynamic analysis is not suited for all use-cases. Specifically, the inherent lack of soundness guarantees for dynamic analysis could lead to too many false negatives in rarely executed code paths. Likewise, in performance-critical environments, the runtime or memory overhead of performing dynamic analysis may not be acceptable.

Voss and Sarkar [22] describe a promise ownership policy in which ownership of a promise by a task denotes the task's responsibility to fulfill the promise or transfer its ownership. This policy is utilized to dynamically detect two classes of bugs: omitted writes and deadlocks. Our static promise ownership model draws on this design, encoding the same ownership relation as a compile-time property. In doing so, we are able to translate the dynamic detection of omitted writes employed by Voss and Sarkar into a compile-time guarantee – precluding false negatives. Further, although the algorithm described by Voss and Sarkar requires syntactic annotations indicating the transfer of promise ownership between tasks, our type system statically infers the same information. As such, our type checking algorithm makes it possible to remove these annotations, thus lowering the user cost of enabling online deadlock detection for programs written in our language.

Two major camps currently exist for static analysis. On the one hand, many static analysis tools are built-in to compilers, often in the form of type systems. These tools generally offer a streamlined approach to analyzing programs as a result of their deep integration into the language. On the other hand, third-party static analysis tools often exist as standalone programs or frameworks meant to augment the guarantees of the language itself. These are often able to provide a valuable addition to large/existing code bases, in that these analyses can easily be retrofit into existing environments. At the same time, because of their nature as language extensions, they may only be able to analyze a subset of programs (leading to situations where errors can leak in through dependencies).

In terms of static analysis for concurrent programs, much of the research has focused on the use of advanced type systems. Boyapati et al. [3] present an extension to Java that encodes a partial ordering of locks using ownership types. This work provides promising results, including preventing all data-races and deadlocks. This work predates the widespread adoption of promises as a concurrency abstraction, and thus does not offer a similarly high-level abstraction for writing concurrent code. The need for annotations also complicates adoption for such a system. Westbrook et al. [25] introduce an extension to Java utilizing fractional permissions to statically prevent data races. This provides analysis using only minor modifications over normal Java programs and uses gradual typing to facilitate an easier translation process. This differs from our work in that it does not allow for the compile-time detection of the promise-based bugs our language prevents. Carbone and Montesi [4] present a deadlock-free type system based on multiparty session types. This also provides promising results, as the language is able to statically prevent a large class of bugs. As with our own work, this language is not based on existing mainstream programming languages. In contrast to our language, however, programming begins with a global specification of the program's behavior. This can then allow for projection of the global protocol to various programming languages to implement the program's business logic, but due to the development methodology requires a new approach to designing concurrent programs.

Ábrahám et al. [27] introduce a programming language that uses affine types for promises. This is somewhat similar to our own approach of using linear types, but does not require the programmer to fulfill all promises along each execution path (because affine types allow for weakening, i.e. an affine variable can be left unused). This prevents the language from statically identifying omitted writes, and by extension means that programs without any deadlocks can still get stuck. The language also disallows promises contained in heap-allocated data, a restriction which we have loosened in our type system. A final difference is that we introduce a deterministic and efficient type checking algorithm for our type system, which could aid in the implementation of such a type system.

Niehren et al. [16] present a linear type system and semantics for a lambda calculus with promises, $\lambda(fut)$. We follow a similar model in describing a type system that makes promise handles linear. Both systems guarantee similar properties about the safety of promises, such as all promises being fulfilled exactly once. However, $\lambda(fut)$'s linear type system is not intended to be used for programming. It is a proof system that one may apply to an existing program to verify correctness properties. This fundamentally differs from our language, in which the use of linear types in the program itself enables the compiler to immediately verify these correctness properties and enables users to build on language-defined abstractions using their safety guarantees. Beyond that, by diverging from the λ-calculus basis of $\lambda(fut)$, we provide a language more akin to mainstream programming languages. Consequently, we believe that we have designed a linear type system that is more intuitive for users. In practice, this means that users are better able to translate conventional code without having to think hard about how to represent it. Additionally, our introduction of a decidable and efficient type checking algorithm creates a simpler path for adoption.

As compared to related works using type systems to prevent concurrency bugs [3, 25, 4, 27, 16], we believe that our language strikes a desirable balance between powerful high-level abstractions, an intuitive programming model, and strong compile-time guarantees. Due to the basis of promises as the core concurrency primitive, we believe that translation of existing promise-based programs to our language would be relatively straight-forward. Based on the familiar programming model (as compared to other general purpose languages) and the lack of complex typing rules/annotations, we also believe that our type system could be easier to

pick up than similar languages. This is especially true with the introduction of substructural typing into mainstream programming languages via Rust, which has shown that similar type systems can be easily employed for general-purpose programming tasks [24].

6.2 Semantics for Concurrent Programs

Various works have explored formalizing the operational semantics of concurrent and/or promise-based programs [15, 13, 16, 25]. Though other works have similar goals in defining their semantics, we find that none mapped perfectly to the goals of this language.

The semantics of $\lambda(fut)$ share many similarities with our own semantics [16]. For example, our semantics models tasks and parallelism in a similar way to $\lambda(fut)$ and builds on similar concurrency constructs, such as promises and forking. In contrast, we believe that our semantics better fit the characteristics of promises in most contemporary programming languages. Specifically, $\lambda(fut)$'s semantics perform promise reads implicitly and only by need: if the value of a promise is never used, then the program will not await the promise before continuing. Such a system could lead to confusing semantics for users unfamiliar with non-strict evaluation and, more importantly, cannot be retrofit onto languages with strict evaluation schemes such as C++, Java, or JavaScript. Thus, we believe that our semantics serve a more appropriate basis for implementation in popular programming languages.

Lee and Palsberg [13] present operational semantics for Featherweight X10, which models threads as a tree of forked tasks. This design fits very well into our work and we drew upon these semantics as inspiration. In contrast to the similar operational semantics of Featherweight X10, however, our rules allow for the free use of async in any expression (via our use of contextual semantics to substitute threads in the tree). This is advantageous in that it generalizes the language so that spawning asynchronous tasks may occur from any expression without restricting us to a linear instruction-based program. As an example, one could not spawn an asynchronous task as part of the condition for an if expression or the right-hand side of a let expression in Featherweight X10.

Westbrook et al. [25] present another operational semantics for concurrent programs. The operational semantics introduces special tracking of heap values and their associated permissions. This design was inspirational in terms of tracking promises and their owners in our operational semantics. However, these semantics build heavily on notions such as permissions, which were not included in our language. Additionally, both the semantics of Featherweight X10 [13] and Westbrook et al. [25] are based on async-finish parallelism, which did not map well to our language due to the lack of a finish construct.

Madsen et al. [15] focus more heavily on promise-based concurrency, defining a language λ_p that formalizes the semantics of JavaScript's promises. Our semantics share many similarities with those of λ_p, such as drawing on a similar model for tracking promise states and describing many of the same classes of promise-based bugs using our semantics. Due to the single-threaded nature of JavaScript, however, JavaScript-style promises rely on *reactions* to events (such as fulfillment or rejection of a promise) rather than distinct, parallel threads of execution. Since our language is inherently parallel and does not feature promise reactions, we could not reuse the same semantics to describe all possible programs in our language.

7 Conclusion & Future Work

Promises are a powerful structured concurrency primitive, which are increasingly being used in modern programming languages. While they represent a huge step forward from

unstructured concurrency, promises can still contribute to their own fair share of bugs. To address this:

1. We have introduced a complete language featuring a novel type system and operational semantics for promise-based programming.
2. Utilizing a linear type system in which promises must be fulfilled exactly once, our language precludes several classes of promise-based bugs – omitted writes in terminating programs, double writes, and unowned writes – with no runtime overhead.
3. Further, by integrating the results directly into the language rather than an external proof system, the language offers the advantages of first-class support for this style of programming. For example, as in other advanced type systems (e.g. Rust's substructural type system [24]), all libraries included in a program satisfy the language's guarantees using no extra instrumentation or annotations. This is in contrast to gradual typing systems – such as Flow [5] and TypeScript [19] – that can only guarantee safety properties for the subset of code that features the proper annotations.
4. With an efficient implementation of the type checker (linearly proportional to the size of the program times the number of variables in scope), this would enable quick verification of large programs. In doing so, a large swathe of promise-related bugs normally only found at runtime will *always* be caught before the program is ever run.

In future work, this language could be extended with several features – such as closures, generics, and arrays – in order to incorporate its design into existing languages like Java or C++. Further, developing a gradual type system based on our approach may serve to streamline integration into gradually typed languages such as TypeScript. Due to the reliance on promises as a core abstraction in JavaScript and TypeScript, this type system could have far reaching effects in the JavaScript/TypeScript ecosystem. Likewise, Haskell could serve as an interesting target for implementation with its recent addition of a linear typing extension.

The role that a type system with owned promises can serve in statically detecting deadlocks remains an open question. Through the availability of more information on promise ownership and thread relationships at compile-time, it may be possible to identify deadlocks early via minor extensions to the type system. Finally, while data race freedom has not been proven for the language, we believe that data races are likely impossible given our design.

We believe that our language provides a strong foundation for designing safer promise abstractions in both novel and existing programming languages. Due to a decidable and efficient type checking algorithm, we believe that the required analysis can be performed at sufficient speeds for real-world programs. Further, we believe that accessing these new guarantees is not overly burdensome to users, as no extra annotations are required to enforce promise ownership. With a similar programming model to promises in C++, we believe that our language provides the proper abstractions to write real-world code. Finally, we believe that, with minimal additions to our type system, it is feasible to extend various *existing* languages in order to adopt the same guarantees.

References

1 Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding broken promises in asynchronous javascript programs. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276532.

2 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158093.

3 Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, page 211–230, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/582419.582440.

4 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: Multiparty asynchronous global programming. *SIGPLAN Not.*, 48(1):263–274, January 2013. doi:10.1145/2480359.2429101.

5 Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for JavaScript. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi:10.1145/3133872.

6 Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 332–343, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2970276.2970347.

7 Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

8 R. E. Fairley. Tutorial: Static analysis and dynamic testing of computer software. *Computer*, 11(4):14–23, 1978. doi:10.1109/C-M.1978.218132.

9 Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, and Huu-Phuc Vo. Forward to a promising future. In Giovanna Di Marzo Serugendo and Michele Loreti, editors, *Coordination Models and Languages*, pages 162–180, Cham, 2018. Springer International Publishing.

10 Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen, and Tobias Wrigstad. Godot: All the Benefits of Implicit and Explicit Futures. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:28, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2019.2.

11 Richard C. Holt. Some deadlock properties of computer systems. *ACM Comput. Surv.*, 4(3):179–196, 1972. doi:10.1145/356603.356607.

12 ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*, pages 1391–1407. International Organization for Standardization, Geneva, Switzerland, fifth edition, 2017. URL: https://www.iso.org/standard/68564.html.

13 Jonathan K. Lee and Jens Palsberg. Featherweight X10: A core calculus for async-finish parallelism. *SIGPLAN Not.*, 45(5):25–36, 2010. doi:10.1145/1837853.1693459.

14 B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.*, 23(7):260–267, 1988. doi:10.1145/960116.54016.

15 Magnus Madsen, Ondřej Lhoták, and Frank Tip. A model for reasoning about JavaScript promises. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017. doi:10.1145/3133910.

16 J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, 2006. doi:10.1016/j.tcs.2006.08.016.

17 Jihyun Park, Byoungju Choi, and Seungyeun Jang. Dynamic analysis method for concurrency bugs in multi-process/multi-thread environments. *International Journal of Parallel Programming*, 48, December 2020. doi:10.1007/s10766-020-00661-3.

18 Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 531–542, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2254064.2254127.

19 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-*

SIGACT Symposium on Principles of Programming Languages, POPL '15, page 167–180, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2676726.2676971.

20 Justin Slepak. Notes on substructural types. URL: http://www.ccs.neu.edu/~jrslepak/substruct-notes.pdf.

21 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bharagavan, and Jean Yang. Secure distributed programming with value-dependent types. Technical Report MSR-TR-2011-37, Microsoft Research, March 2011. This is an extended version of the conference paper (ICFP '11) with the same title. A final version of this full technical report is forthcoming. URL: https://www.microsoft.com/en-us/research/publication/secure-distributed-programming-with-value-dependent-types/.

22 Caleb Voss and Vivek Sarkar. An ownership policy and deadlock detector for promises, 2021. arXiv:2101.01312.

23 P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.

24 Aaron Weiss, Olek Gierczak, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The essence of Rust, 2020. arXiv:1903.00982v3.

25 Edwin Westbrook, Jisheng Zhao, Zoran Budimlić, and Vivek Sarkar. Practical permissions for race-free parallelism. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 614–639, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

26 Dengping Zhu. *To Memory Safety through Proofs*. PhD thesis, Boston University, USA, 2006.

27 Erika Ábrahám, Immo Grabe, Andreas Grüner, and Martin Steffen. Behavioral interface description of an object-oriented language with futures and promises. *The Journal of Logic and Algebraic Programming*, 78(7):491–518, 2009. The 19th Nordic Workshop on Programming Theory (NWPT 2007). doi:10.1016/j.jlap.2009.01.001.

Lifted Static Analysis of Dynamic Program Families by Abstract Interpretation

Aleksandar S. Dimovski
Mother Teresa University, Skopje, North Macedonia

Sven Apel
Saarland University, Saarland Informatics Campus, 66123 Saarbrücken, Germany

Abstract

Program families (software product lines) are increasingly adopted by industry for building families of related software systems. A program family offers a set of *features* (configured options) to control the presence and absence of software functionality. Features in program families are often assigned at compile-time, so their values can only be *read* at run-time. However, today many program families and application domains demand run-time adaptation, reconfiguration, and post-deployment tuning. Dynamic program families (dynamic software product lines) have emerged as an attempt to handle variability at run-time. Features in dynamic program families can be controlled by ordinary program variables, so *reads* and *writes* to them may happen at run-time.

Recently, a decision tree lifted domain for analyzing traditional program families with numerical features has been proposed, in which decision nodes contain linear constraints defined over numerical features and leaf nodes contain analysis properties defined over program variables. Decision nodes partition the configuration space of possible feature values, while leaf nodes provide analysis information corresponding to each partition of the configuration space. As features are statically assigned at compile-time, decision nodes can be added, modified, and deleted only when analyzing read accesses of features. In this work, we extend the decision tree lifted domain so that it can be used to efficiently analyze dynamic program families with numerical features. Since features can now be changed at run-time, decision nodes can be modified when handling read and write accesses of feature variables. For this purpose, we define extended transfer functions for assignments and tests as well as a special widening operator to ensure termination of the lifted analysis. To illustrate the potential of this approach, we have implemented a lifted static analyzer, called DSPLNum^{2}Analyzer, for inferring numerical invariants of dynamic program families written in C. An empirical evaluation on benchmarks from SV-COMP indicates that our tool is effective and provides a flexible way of adjusting the precision/cost ratio in static analysis of dynamic program families.

2012 ACM Subject Classification Software and its engineering → Software product lines; Theory of computation → Abstraction; Software and its engineering → Software functional properties

Keywords and phrases Dynamic program families, Static analysis, Abstract interpretation, Decision tree lifted domain

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.14

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.6`

1 Introduction

A *program family* (software product line) is a set of similar programs, called *variants*, that is built from a common code base [39]. The variants of a program family can be distinguished in terms of *features*, which describe the commonalities and variability between the variants. Program families are commonly seen in the development of commercial embedded and critical system domains, such as cars, phones, avionics, medicine, robotics, etc. [1]. There are several techniques for implementing program families. Often traditional program families [11] support static feature binding and require to know the values of features at compile-time. For example, `#if` directives from the C preprocessor `CPP` represent the most common

implementation mechanism in practice [34]. At compile-time, a variant is derived by assigning concrete values to a set of features relevant for it, and only then is this variant compiled or interpreted. However, in an increasingly dynamic world, the increasing need for adaptive software demands highly configurable and adaptive variability mechanisms, many of them managed at run-time. Recent development approaches such as dynamic program families (dynamic software product lines) [29, 28, 41, 7] support dynamic feature binding, and so features can be assigned at run-time. This provides high flexibility to tailor a variant with respect to available resources and user preferences on demand. Dynamic binding is often necessary in long-running systems that cannot be stopped but have to adapt to changing requirements [27]. For example, for a mobile device, we can decide at run-time which values of features are actually required according to the location of the device. Hence, a dynamic program family adapts to dynamically changing requirements by reconfiguring itself, which may result in an infinite configuration process [10].

In this paper, we devise an approach to perform static analysis by abstract interpretation of dynamic program families. *Abstract interpretation* [12, 38] is a powerful framework for approximating the semantics of programs. It provides static analysis techniques that analyze the program's source code directly and without intervention at some level of abstraction. The obtained static analyses are sound (all reported correct programs are indeed correct) and efficient (with a good trade-off between precision and cost). However, static analysis of program families is harder than static analysis of single programs, because the number of possible variants can be very large (often huge) in practice. Recently, researchers have addressed this problem by designing aggregate *lifted* (family-based) *static analyses* [5, 36, 47], which analyze all variants of the family simultaneously in a single run. These techniques take as input the common code base, which encodes all variants of a program family, and produce precise analysis results for all variants. Lifted static analysis by abstract interpretation of traditional (static) program families with numerical features has been introduced recently [21]. The elements of the lifted abstract domain are *decision trees*, in which the decision nodes are labelled with linear constraints over numerical features, whereas the leaf nodes belong to a single-program analysis domain. The decision trees recursively partition the space of configurations (i.e., the space of possible combinations of feature values), whereas the program properties at the leaves provide analysis information corresponding to each partition, i.e. to the variants (configurations) that satisfy the constraints along the path to the given leaf node. Since features are statically bound at compile-time and only appear in presence conditions of `#if` directives, new decision nodes can only be added by feature-based presence conditions (at `#if` directives), and existing decision nodes can be removed when merging the corresponding control flows again. The fundamental limitation of this decision tree lifted domain [21] (as well as other lifted domains [4, 36, 47]) is that it cannot handle dynamically bound features that can be changed at run-time.

To improve over the state-of-the-art, we devise a novel decision tree lifted domain for analyzing dynamic program families with numerical features. Since features can now be dynamically reconfigured and bound at run-time, linear constraints over features that occur in decision nodes can be dynamically changed during the analysis. This requires extended transfer functions for assignments and tests that can freely modify decision nodes and leafs. Moreover, we need a special widening operator applied on linear constraints in decision nodes as well as on analysis properties in leaf nodes to ensure that we obtain finite decision trees. This way, we minimize the cost of the lifted analysis and ensure its termination.

The resulting decision tree lifted domain is parametric in the choice of the numerical domain that underlies the linear constraints over numerical features labelling decision nodes, and the choice of the single-program analysis domain for leaf nodes. In our implementation,

we also use numerical domains for leaf nodes, which encode linear constraints over both program and feature variables. We use well-known numerical domains, including intervals [12], octagons [37], polyhedra [16], from the APRON library [33], to obtain a concrete decision tree-based implementation of the lifted abstract domain. To demonstrate the feasibility of our approach, we have implemented a *lifted analysis* of dynamic program families written in C for the automatic inference of numerical invariants. Our tool, called DSPLNum^{2}Analyzer[1], computes a set of possible numerical invariants, which represent linear constraints over program and feature variables. We can use the implemented lifted static analyzer to check invariance properties of dynamic program families in C, such as assertions, buffer overflows, null pointer references, division by zero, etc. [14].

Since features behave as ordinary program variables in dynamic program families, they can be also analyzed using off-the-shelf single-program analyzers. For example, we can use numerical abstract domains from the APRON library [33] for analyzing dynamic program families. However, these domains infer a conjunction of linear constraints over variables to record the information of all possible values of variables and relationships between them. The absence of disjunctions may result in rough approximations and very weak analysis results, which may lead to imprecisions and the failure of showing the required program properties. The decision tree lifted domain proposed here overcomes these limitations of standard single-program analysis domains by adding weak forms of disjunctions arising from feature-based program constructs. The elements of the decision tree lifted domain partition the space of possible values of features inducing disjunctions into the leaf domain.

In summary, we make several contributions:

- We propose a new parameterized decision tree lifted domain suited for handling program families with dynamically bound features.
- We develop a lifted static analyzer, DSPLNum^{2}Analyzer, in which the lifted domain is instantiated to numerical domains from the APRON library.
- We evaluate our approach for lifted static analysis of dynamic program families written in C. We compare (precision and time) performances of our decision tree-based approach with the single-program analysis approach; and we show their concrete application in assertion checking. Our lifted analysis provides an acceptable precision/cost tradeoff: we obtain invariants with a higher degree of precision within a reasonable amount of time than when using single-program analysis.

2 Motivating Example

We now illustrate the decision tree lifted domain through several motivating examples. The code base of the program family sFAMILY is given in Fig. 1. sFAMILY contains one numerical feature `A` whose domain is $[0, 99] = \{0, 1, \ldots, 99\}$. Thus, there are a hundred valid configurations $\mathbb{K} = \{(\texttt{A}=0), (\texttt{A}=1), \ldots, (\texttt{A}=99)\}$. The code of sFAMILY contains one `#if` directive that changes the current value of program variable `y` depending on how feature `A` is set at compile-time. For each configuration from $\mathbb{K}$, a variant (single program) can be generated by appropriately resolving the `#if` directive. For example, the variant corresponding to configuration $(\texttt{A}=0)$ will have the assignment `y := y+1` included in location ③, whereas the variant corresponding to configuration $(\texttt{A}=10)$ will have the assignment `y := y-1` included in location ③.

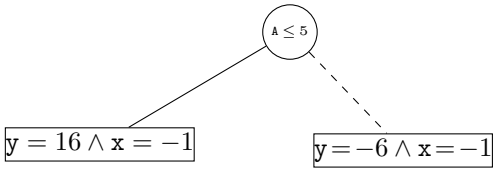
[1] Num^2 in the name of the tool refers to its ability to both handle NUMerical features and to perform NUMerical client analysis of dynamic program families (DSPLs).

```
① int x := 10, y := 5;
② while (x  ≥ 0) {
③     #if (A ≤ 5) y := y+1;
④         #else y := y-1; #endif
⑤     x := x-1;
⑥ }
```

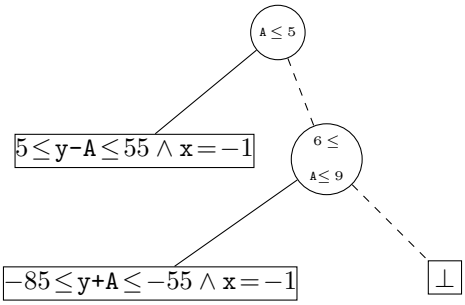
Figure 1 Program family sFAMILY.

```
① int x := 10, y := 5;
② A := [0, 9];
③ while (x  ≥ 0) {
④     if (A ≤ 5) then y := y+A;
⑤                else y := y-A;
⑥     x := x-1; }
⑦ if (A ≤ 5) then assert (y ≥ 5);
⑧           else assert (y ≤ −60);
```

Figure 2 Dynamic program family dFAMILY.



(a) sFAMILY.



(b) dFAMILY.

Figure 3 Inferred decision trees at final program locations (solid edges = true, dashed edges = false).

Assume that we want to perform *lifted polyhedra analysis* of sFAMILY using the decision tree lifted domain introduced in [21]. The decision tree inferred at the final location of sFAMILY is shown in Fig. 3a. Notice that inner decision nodes (resp., leaves) of the decision tree in Fig. 3a are labeled with *Polyhedra* linear constraints over feature `A` (resp., over program variables `x` and `y`). The edges of decision trees are labeled with the truth value of the decision on the parent node; we use solid edges for true (i.e. the constraint in the parent node is satisfied) and dashed edges for false (i.e. the negation of the constraint in the parent node is satisfied). We observe that decision trees offer good possibilities for sharing and interaction between analysis properties corresponding to different configurations, and so they provide compact representation of lifted analysis elements. For example, the decision tree in Fig. 3a shows that when $(\mathtt{A} \leq 5)$ the shared property in the final location is $(\mathtt{y} = 16, \mathtt{x} = -1)$, whereas when $(\mathtt{A} > 5)$ the shared property is $(\mathtt{y} = -6, \mathtt{x} = -1)$. Hence, the decision tree-based approach uses only two leaves (program properties), whereas the brute force enumeration approach that analyzes all variants one by one will use a hundred program properties. This ability for sharing is the key motivation behind the usage of decision trees in lifted analysis.

Consider the code base of the dynamic program family dFAMILY in Fig. 2. Similarly to sFAMILY, dFAMILY contains one feature `A` with domain $[0, 99]$. However, feature `A` in sFAMILY can only be read and occurs only in presence conditions of `#if`-s. In contrast, feature `A` in dFAMILY can also be assigned and occurs freely in the code as any other program variable (see locations ②, ④, ⑤, and ⑦). To perform *lifted polyhedra analysis* of dFAMILY, we need to extend the decision tree lifted domain for traditional program families [21], so that it takes into account the new possibilities of features in dynamic program families. The decision tree inferred in program location ⑦ of dFAMILY is depicted in Fig. 3b. It can be written as the following disjunctive property in first order logic:

$$\big(0 \leq \mathtt{A} \leq 5 \wedge 5 \leq \mathtt{y-A} \leq 55 \wedge \mathtt{x} = -1\big) \vee \big(6 \leq \mathtt{A} \leq 9 \wedge -85 \leq \mathtt{y+A} \leq -55 \wedge \mathtt{x} = -1\big) \vee \big(9 < \mathtt{A} \leq 99 \wedge \bot\big)$$

This invariant successfully confirms the validity of the given assertion. Note that, the leaf node $\bot$ abstracts only the empty set of (concrete) program states and so it describes unreachable program locations. Hence, $\bot$ in Fig. 3b means that the assertion at location ⑦ is unreachable when $(\texttt{A} > 9)$. Also, as decision nodes partition the space of valid configurations $\mathbb{K}$, we implicitly assume the correctness of linear constraints that take into account domains of features. For example, the decision node $(\texttt{A} \leq 5)$ is satisfied when $(\texttt{A} \leq 5) \wedge (0 \leq \texttt{A} \leq 99)$, whereas its negation is satisfied when $(\texttt{A} > 5) \wedge (0 \leq \texttt{A} \leq 99)$. The constraint $(0 \leq \texttt{A} \leq 99)$ represents the domain of A.

Alternatively, dynamic program family DFAMILY can be analyzed using the off-the-shelf (single-program) APRON polyhedra domain [33], such that feature A is considered as an ordinary program variable. In this case, we obtain the invariant: $\texttt{A+y} \leq 66 \wedge \texttt{A-y} \geq -54$ at location ⑦. However, this invariant is not strong enough to establish the validity of the given assertion. This is because the different partitions of the set of valid configurations have different behaviours and this single-program domain do not consider them separately. Therefore, this domain is less precise than the decision tree lifted domain that takes those differences into account.

3 A Language for Dynamic Program Families

Let $\mathbb{F} = \{A_1, \ldots, A_n\}$ be a finite and totaly ordered set of *numerical features* available in a dynamic program family. For each feature $A \in \mathbb{F}$, $\mathrm{dom}(A) \subseteq \mathbb{Z}$ denotes the set of possible values that can be assigned to A. Note that any Boolean feature can be represented as a numerical feature $B \in \mathbb{F}$ with $\mathrm{dom}(B) = \{0, 1\}$, such that 0 means that feature B is disabled while 1 means that B is enabled. An assignment of values to all features represents a *configuration* k, which specifies one *variant* of a program family. It is given as a *valuation function* $k : \mathbb{K} = \mathbb{F} \to \mathbb{Z}$, which is a mapping that assigns a value from $\mathrm{dom}(A)$ to each feature A, i.e. $k(A) \in \mathrm{dom}(A)$ for any $A \in \mathbb{F}$. We assume that only a subset $\mathbb{K}$ of all possible configurations are *valid*. An alternative representation of configurations is based upon propositional formulae. Each configuration $k \in \mathbb{K}$ can be represented by a formula: $(A_1 = k(A_1)) \wedge \ldots \wedge (A_n = k(A_n))$. Given a Boolean feature $B \in \mathbb{F}$, we often abbreviate $(B = 1)$ with formula B and $(B = 0)$ with formula $\neg B$. The set of valid configurations $\mathbb{K}$ can be also represented as a formula: $\vee_{k \in \mathbb{K}} k$.

We consider a simple sequential non-deterministic programming language, which will be used to exemplify our work. The program variables *Var* are statically allocated and the only data type is the set $\mathbb{Z}$ of mathematical integers. To introduce dynamic variability into the language, apart from reading the current values of features, it is possible to write into features. The new statement "A:=*ae*" has a possibility to update the current configuration (variant) $k \in \mathbb{K}$ by assigning a new arithmetic expression *ae* to feature A. This is known as *run-time reconfiguration* [7]. We write $k[A \mapsto n]$ for the updated configuration that is identical to k but feature A is mapped to value n. The syntax of the language is:

$$s ::= \texttt{skip} \mid \texttt{x:=}ae \mid s; s \mid \texttt{if}\ (be)\ \texttt{then}\ s\ \texttt{else}\ s \mid \texttt{while}\ (be)\ \texttt{do}\ s \mid \texttt{A:=}ae,$$
$$ae ::= n \mid [n, n'] \mid \texttt{x} \in \mathit{Var} \mid A \in \mathbb{F} \mid ae \oplus ae,$$
$$be ::= ae \bowtie ae \mid \neg be \mid be \wedge be \mid be \vee be$$

where n ranges over integers $\mathbb{Z}$, $[n, n']$ over integer intervals, x over program variables *Var*, A over numerical features $\mathbb{F}$, and $\oplus \in \{+, -, *, /\}$, $\bowtie \in \{<, \leq, =, \neq\}$. Integer intervals $[n, n']$ denote a random choice of an integer in the interval. The set of all statements s is denoted by *Stm*; the set of all arithmetic expressions *ae* is denoted by *AExp*; the set of all boolean expressions *be* is denoted by *BExp*.

$$
\begin{aligned}
[\![\texttt{skip}]\!]S &= S \\
[\![\texttt{x := } ae]\!]S &= \{\langle \sigma[\texttt{x} \mapsto n], k\rangle \mid \langle \sigma, k\rangle \in S, n \in [\![ae]\!]\langle \sigma, k\rangle\} \\
[\![s_1 \texttt{ ; } s_2]\!]S &= [\![s_2]\!]([\![s_1]\!]S) \\
[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]S &= [\![s_1]\!]\{\langle \sigma, k\rangle \in S \mid \text{true} \in [\![be]\!]\langle \sigma, k\rangle\} \cup \\
&\qquad [\![s_2]\!]\{\langle \sigma, k\rangle \in S \mid \text{false} \in [\![be]\!]\langle \sigma, k\rangle\} \\
[\![\texttt{while } be \texttt{ do } s]\!]S &= \{\langle \sigma, k\rangle \in \operatorname{lfp} \phi \mid \text{false} \in [\![be]\!]\langle \sigma, k\rangle\} \\
\phi(X) &= S \cup [\![s]\!]\{\langle \sigma, k\rangle \in X \mid \text{true} \in [\![be]\!]\langle \sigma, k\rangle\} \\
[\![A \texttt{ := } ae]\!]S &= \{\langle \sigma, k[A \mapsto n]\rangle \mid \langle \sigma, k\rangle \in S, n \in [\![ae]\!]\langle \sigma, k\rangle, k[A \mapsto n] \in \mathbb{K}\}
\end{aligned}
$$

Figure 4 Invariance semantics $[\![s]\!] : \mathcal{P}(\Sigma \times \mathbb{K}) \to \mathcal{P}(\Sigma \times \mathbb{K})$.

Semantics

We now define the semantics of a dynamic program family. A *store* $\sigma : \Sigma = \mathit{Var} \to \mathbb{Z}$ is a mapping from program variables to values, whereas a *configuration* $k : \mathbb{K} = \mathbb{F} \to \mathbb{Z}$ is a mapping from numerical features to values. A *program state* $s = \langle \sigma, k\rangle : \Sigma \times \mathbb{K}$ is a pair consisting of a store $\sigma \in \Sigma$ and a configuration $k \in \mathbb{K}$. The semantics of arithmetic expressions $[\![ae]\!] : \Sigma \times \mathbb{K} \to \mathcal{P}(\mathbb{Z})$ is the set of possible values for expression ae in a given state. It is defined by induction on ae as a function from a store and a configuration to a set of values:

$$
\begin{aligned}
&[\![n]\!]\langle \sigma, k\rangle = \{n\},\ [\![[n, n']]\!]\langle \sigma, k\rangle = \{n, \ldots, n'\},\ [\![\texttt{x}]\!]\langle \sigma, k\rangle = \{\sigma(\texttt{x})\}, \\
&[\![A]\!]\langle \sigma, k\rangle = \{k(A)\},\ [\![ae_0 \oplus ae_1]\!]\langle \sigma, k\rangle = \{n_0 \oplus n_1 \mid n_0 \in [\![ae_0]\!]\langle \sigma, k\rangle, n_1 \in [\![ae_1]\!]\langle \sigma, k\rangle\}
\end{aligned}
$$

Similarly, the semantics of boolean expressions $[\![be]\!] : \Sigma \times \mathbb{K} \to \mathcal{P}(\{\text{true}, \text{false}\})$ is the set of possible truth values for expression be in a given state.

$$
\begin{aligned}
&[\![ae_0 \bowtie ae_1]\!]\langle \sigma, k\rangle = \{n_0 \bowtie n_1 \mid n_0 \in [\![ae_0]\!]\langle \sigma, k\rangle, n_1 \in [\![ae_1]\!]\langle \sigma, k\rangle\} \\
&[\![\neg be]\!]\langle \sigma, k\rangle = \{\neg t \mid t \in [\![be]\!]\langle \sigma, k\rangle\}, \\
&[\![be_0 \wedge be_1]\!]\langle \sigma, k\rangle = \{t_0 \wedge t_1 \mid t_0 \in [\![be_0]\!]\langle \sigma, k\rangle, t_1 \in [\![be_1]\!]\langle \sigma, k\rangle\} \\
&[\![be_0 \vee be_1]\!]\langle \sigma, k\rangle = \{t_0 \vee t_1 \mid t_0 \in [\![be_0]\!]\langle \sigma, k\rangle, t_1 \in [\![be_1]\!]\langle \sigma, k\rangle\}
\end{aligned}
$$

We define an *invariance semantics* [12, 38] on the complete lattice $\langle \mathcal{P}(\Sigma \times \mathbb{K}), \subseteq, \cup, \cap, \emptyset, \Sigma \times \mathbb{K}\rangle$ by induction on the syntax of programs. It works on *sets of states*, so the property of interest is the possible sets of stores and configurations that may arise at each program location. In Fig. 4, we define the invariance semantics $[\![s]\!] : \mathcal{P}(\Sigma \times \mathbb{K}) \to \mathcal{P}(\Sigma \times \mathbb{K})$ of each program statement. The states resulting from the invariance semantics are built forward: each function $[\![s]\!]$ takes as input a set of states (i.e. pairs of stores and configurations) $S \in \mathcal{P}(\Sigma \times \mathbb{K})$ and outputs the set of possible states at the final location of the statement. The operation $k[A \mapsto n]$ (resp., $\sigma[\texttt{x} \mapsto n]$) is used to update a configuration from $\mathbb{K}$ (resp., a store from Σ). Note that a `while` statement is given in a standard fixed-point formulation [12], where the fixed-point functional $\phi : \mathcal{P}(\Sigma \times \mathbb{K}) \to \mathcal{P}(\Sigma \times \mathbb{K})$ accumulates the possible states after another `while` iteration from a given set of states X.

However, the invariance semantics $[\![s]\!]$ is not computable since our language is Turing complete. In the following, we present sound decidable abstractions of $[\![s]\!]$ by means of decision tree-based abstract domains.

4 Decision Trees Lifted Domain

Lifted analyses are designed by *lifting* existing single-program analyses to work on program families, rather than on individual programs. Lifted analysis for traditional program families introduced in [21] relies on a *decision tree lifted domain*. The leaf nodes of decision trees belong to an existing single-program analysis domain, and are separated by linear constraints over numerical features, organized in decision nodes. In Section 4.1, we first recall basic elements of the decision tree lifted domain [21] that can be reused for dynamic program families. Then, in Section 4.2 we consider extended transfer functions for assignments and tests when features can freely occur in them, whereas in Section 4.3 we define the extrapolation widening operator for this lifted domain. Finally, we define the abstract invariance semantics based on this domain and show its soundness in Section 4.4.

4.1 Basic elements

Abstract domain for leaf nodes

We assume that a single-program numerical domain $\mathbb{D}$ defined over a set of variables V is equipped with sound operators for concretization $\gamma_{\mathbb{D}}$, ordering $\sqsubseteq_{\mathbb{D}}$, join $\sqcup_{\mathbb{D}}$, meet $\sqcap_{\mathbb{D}}$, the least element (called bottom) $\bot_{\mathbb{D}}$, the greatest element (called top) $\top_{\mathbb{D}}$, widening $\nabla_{\mathbb{D}}$, and narrowing $\triangle_{\mathbb{D}}$, as well as sound transfer functions for tests (boolean expressions) $\text{FILTER}_{\mathbb{D}}$ and forward assignments $\text{ASSIGN}_{\mathbb{D}}$. The domain $\mathbb{D}$ employs data structures and algorithms specific to the shape of invariants (analysis properties) it represents and manipulates. More specifically, the concretization function $\gamma_{\mathbb{D}}$ assigns a concrete meaning to each element in $\mathbb{D}$, ordering $\sqsubseteq_{\mathbb{D}}$ conveys the idea of approximation since some analysis results may be coarser than some other results, whereas join $\sqcup_{\mathbb{D}}$ and meet $\sqcap_{\mathbb{D}}$ convey the idea of convergence since a new abstract element is computed when merging control flows. To analyze loops effectively and efficiently, the convergence acceleration operators such as widening $\nabla_{\mathbb{D}}$ and narrowing $\triangle_{\mathbb{D}}$ are used. Transfer functions give abstract semantics of expressions and statements. Hence, $\text{ASSIGN}_{\mathbb{D}}(d : \mathbb{D}, \texttt{x:=}ae : \mathit{Stm})$ returns an updated version of d by abstractly evaluating $\texttt{x:=}ae$ in it, whereas $\text{FILTER}_{\mathbb{D}}(d : \mathbb{D}, be : \mathit{BExp})$ returns an abstract element from $\mathbb{D}$ obtained by restricting d to satisfy test be. In practice, the domain $\mathbb{D}$ will be instantiated with some of the known numerical domains, such as Intervals $\langle I, \sqsubseteq_I \rangle$ [12], Octagons $\langle O, \sqsubseteq_O \rangle$ [46], and Polyhedra $\langle P, \sqsubseteq_P \rangle$ [16]. The elements of I are intervals of the form: $\pm x \geq \beta$, where $x \in V, \beta \in \mathbb{Z}$; the elements of O are conjunctions of octagonal constraints of the form $\pm x_1 \pm x_2 \geq \beta$, where $x_1, x_2 \in V, \beta \in \mathbb{Z}$; while the elements of P are conjunctions of polyhedral constraints of the form $\alpha_1 x_1 + \ldots + \alpha_k x_k + \beta \geq 0$, where $x_1, \ldots x_k \in V, \alpha_1, \ldots, \alpha_k, \beta \in \mathbb{Z}$.

We will sometimes write $\mathbb{D}_V$ to explicitly denote the set of variables V over which domain $\mathbb{D}$ is defined. In this work, we use domains $\mathbb{D}_{Var \cup \mathbb{F}}$ for leaf nodes of decision trees that are defined over both program and feature variables. The abstraction for numerical domains $\langle \mathbb{D}_{Var \cup \mathbb{F}}, \sqsubseteq_{\mathbb{D}} \rangle$ is formally defined by the concretization-based abstraction $\langle \mathcal{P}(\Sigma \times \mathbb{K}), \subseteq \rangle \xleftarrow{\gamma_{\mathbb{D}}} \langle \mathbb{D}_{Var \cup \mathbb{F}}, \sqsubseteq_{\mathbb{D}} \rangle$. We refer to [38] for a more detailed discussion of the definition of $\gamma_{\mathbb{D}}$ as well as other abstract operations and transfer functions for Intervals, Octagons, and Polyhedra.

Abstract domain for decision nodes

We introduce a family of abstract domains for linear constraints $\mathbb{C}_{\mathbb{D}}$ defined over features $\mathbb{F}$, which are parameterized by any of the numerical domains $\mathbb{D}$ (intervals I, octagons O, polyhedra P). For example, the finite set of *polyhedral constraints* is $\mathbb{C}_P = \{\alpha_1 A_1 + \ldots + \alpha_k A_k + \beta \geq 0 \mid A_1, \ldots A_k \in \mathbb{F}, \alpha_1, \ldots, \alpha_k, \beta \in \mathbb{Z}, \gcd(|\alpha_1|, \ldots, |\alpha_k|, |\beta|) = 1\}$. The finite set

$\mathbb{C}_\mathbb{D}$ of linear constraints over features $\mathbb{F}$ is constructed by the underlying numerical domain $\langle \mathbb{D}, \sqsubseteq_\mathbb{D} \rangle$ using the Galois connection $\langle \mathcal{P}(\mathbb{C}_\mathbb{D}), \sqsubseteq_\mathbb{D} \rangle \underset{\alpha_{\mathbb{C}_\mathbb{D}}}{\overset{\gamma_{\mathbb{C}_\mathbb{D}}}{\leftrightarrows}} \langle \mathbb{D}, \sqsubseteq_\mathbb{D} \rangle$, where $\mathcal{P}(\mathbb{C}_\mathbb{D})$ is the power set of $\mathbb{C}_\mathbb{D}$. The concretization function $\gamma_{\mathbb{C}_\mathbb{D}} : \mathbb{D} \to \mathcal{P}(\mathbb{C}_\mathbb{D})$ maps a conjunction of constraints from $\mathbb{D}$ to a finite set of constraints in $\mathcal{P}(\mathbb{C}_\mathbb{D})$.

The domain of decision nodes is $\mathbb{C}_\mathbb{D}$. We assume the set of features $\mathbb{F} = \{A_1, \ldots, A_n\}$ to be totally ordered, such that the ordering is $A_1 > \ldots > A_n$. We impose a total order $<_{\mathbb{C}_\mathbb{D}}$ on $\mathbb{C}_\mathbb{D}$ to be the lexicographic order on the coefficients $\alpha_1, \ldots, \alpha_n$ and constant α_{n+1} of the linear constraints, such that:

$$\begin{aligned}&(\alpha_1 \cdot A_1 + \ldots + \alpha_n \cdot A_n + \alpha_{n+1} \geq 0) \;<_{\mathbb{C}_\mathbb{D}}\; (\alpha'_1 \cdot A_1 + \ldots + \alpha'_n \cdot A_n + \alpha'_{n+1} \geq 0)\\ &\iff\; \exists j > 0. \forall i < j. (\alpha_i = \alpha'_i) \wedge (\alpha_j < \alpha'_j)\end{aligned}$$

The negation of linear constraints is formed as: $\neg(\alpha_1 A_1 + \ldots \alpha_n A_n + \beta \geq 0) = -\alpha_1 A_1 - \ldots - \alpha_n A_n - \beta - 1 \geq 0$. For example, the negation of $A - 3 \geq 0$ is $-A + 2 \geq 0$. To ensure canonical representation of decision trees, a linear constraint c and its negation $\neg c$ cannot both appear as decision nodes. Thus, we only keep the largest constraint with respect to $<_{C_\mathbb{D}}$ between c and $\neg c$.

Abstract domain for decision trees

A *decision tree* $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}_\mathbb{F}}, \mathbb{D}_{Var \cup \mathbb{F}})$ over the sets $\mathbb{C}_{\mathbb{D}_\mathbb{F}}$ of linear constraints defined over $\mathbb{F}$ and the leaf abstract domain $\mathbb{D}_{Var \cup \mathbb{F}}$ defined over $Var \cup \mathbb{F}$ is: either a leaf node $\ll d \gg$ with $d \in \mathbb{D}_{Var \cup \mathbb{F}}$, or $[\![c : tl, tr]\!]$, where $c \in \mathbb{C}_{\mathbb{D}_\mathbb{F}}$ (denoted by $t.c$) is *the smallest constraint* with respect to $<_{\mathbb{C}_\mathbb{D}}$ appearing in the tree t, tl (denoted by $t.l$) is the left subtree of t representing its *true branch*, and tr (denoted by $t.r$) is the right subtree of t representing its *false branch*. The path along a decision tree establishes the set of configurations (those that satisfy the encountered constraints), and the leaf nodes represent the analysis properties for the corresponding configurations.

▶ **Example 1.** The following two decision trees t_1 and t_2 have decision and leaf nodes labelled with polyhedral linear constraints defined over numerical feature `A` with domain $\mathbb{Z}$ and over integer program variable y, respectively:

$$t_1 = [\![\mathtt{A} \geq 4 : \ll [y \geq 2] \gg, \ll [y = 0] \gg]\!],\; t_2 = [\![\mathtt{A} \geq 2 : \ll [y \geq 0] \gg, \ll [y \leq 0] \gg]\!]$$ ⌟

Abstract Operations

We define the following concretization-based abstraction $\langle \mathcal{P}(\Sigma \times \mathbb{K}), \subseteq \rangle \xleftarrow{\gamma_\mathbb{T}} \langle \mathbb{T}(\mathbb{C}_\mathbb{D}, \mathbb{D}), \sqsubseteq_\mathbb{T} \rangle$. The *concretization function* $\gamma_\mathbb{T}$ of a decision tree $t \in \mathbb{T}(\mathbb{C}_\mathbb{D}, \mathbb{D})$ returns a set of pairs $\langle \sigma, k \rangle$, such that $\langle \sigma, k \rangle \in \gamma_\mathbb{D}(d)$ and k satisfies the set $C \in \mathcal{P}(\mathbb{C}_\mathbb{D})$ of constraints accumulated along the top-down path to the leaf node $d \in \mathbb{D}$. More formally, the concretization function $\gamma_\mathbb{T}(t) : \mathbb{T}(\mathbb{C}_\mathbb{D}, \mathbb{D}) \to \mathcal{P}(\Sigma \times \mathbb{K})$ is defined as:

$$\gamma_\mathbb{T}(t) = \overline{\gamma}_\mathbb{T}[\mathbb{K}](t)$$

where $\mathbb{K} \in \mathcal{P}(\mathbb{C}_\mathbb{D})$ is the set of configurations, i.e. the set of constraints over $\mathbb{F}$ taking into account the domains of features. Function $\overline{\gamma}_\mathbb{T} : \mathcal{P}(\mathbb{C}_\mathbb{D}) \to \mathbb{T}(\mathbb{C}_\mathbb{D}, \mathbb{D}) \to \mathcal{P}(\Sigma \times \mathbb{K})$ is defined as:

$$\begin{aligned}&\overline{\gamma}_\mathbb{T}[C](\ll d \gg) = \{\langle \sigma, k \rangle \mid \langle \sigma, k \rangle \in \gamma_\mathbb{D}(d), k \models C\},\\ &\overline{\gamma}_\mathbb{T}[C]([\![c : tl, tr]\!]) = \overline{\gamma}_\mathbb{T}[C \cup \{c\}](tl) \;\cup\; \overline{\gamma}_\mathbb{T}[C \cup \{\neg c\}](tr)\end{aligned}$$

Note that $k \models C$ is equivalent with $\alpha_{\mathbb{C}_\mathbb{D}}(\{k\}) \sqsubseteq_\mathbb{D} \alpha_{\mathbb{C}_\mathbb{D}}(C)$, thus we can check $k \models C$ using the abstract operation $\sqsubseteq_\mathbb{D}$ of the numerical domain $\mathbb{D}$.

Algorithm 1 `UNIFICATION`(t_1, t_2, C).

```
1  if isLeaf(t1) ∧ isLeaf(t2) then return (t1, t2);
2  if isLeaf(t1) ∨ (isNode(t1) ∧ isNode(t2) ∧ t2.c <_{C_D} t1.c) then
3      if isRedundant(t2.c, C) then return UNIFICATION(t1, t2.l, C);
4      if isRedundant(¬t2.c, C) then return UNIFICATION(t1, t2.r, C);
5      (l1, l2) = UNIFICATION(t1, t2.l, C ∪ {t2.c});
6      (r1, r2) = UNIFICATION(t1, t2.r, C ∪ {¬t2.c});
7      return ([[t2.c : l1, r1]], [[t2.c : l2, r2]]);
8  if isLeaf(t2) ∨ (isNode(t1) ∧ isNode(t2) ∧ t1.c <_{C_D} t2.c) then
9      if isRedundant(t1.c, C) then return UNIFICATION(t1.l, t2, C);
10     if isRedundant(¬t1.c, C) then return UNIFICATION(t1.r, t2, C);
11     (l1, l2) = UNIFICATION(t1.l, t2, C ∪ {t1.c});
12     (r1, r2) = UNIFICATION(t1.r, t2, C ∪ {¬t1.c});
13     return ([[t1.c : l1, r1]], [[t1.c : l2, r2]]);
14 else
15     if isRedundant(t1.c, C) then return UNIFICATION(t1.l, t2.l, C);
16     if isRedundant(¬t1.c, C) then return UNIFICATION(t1.r, t2.r, C);
17     (l1, l2) = UNIFICATION(t1.l, t2.l, C ∪ {t1.c});
18     (r1, r2) = UNIFICATION(t1.r, t2.r, C ∪ {¬t1.c});
19     return ([[t1.c : l1, r1]], [[t1.c : l2, r2]]);
```

Other binary operations rely on the algorithm for *tree unification* [45] given in Algorithm 1, which finds a common labelling of two trees t_1 and t_2 by forcing them to have the same structure. It accumulates into the set $C \in \mathcal{P}(\mathbb{C}_\mathbb{D})$ (initially equal to $\mathbb{K}$) the linear constraints encountered along the paths of the decision trees possibly adding new constraints as decision nodes (Lines 5–7, Lines 11–13) or removing constraints that are redundant with respect to C (Lines 3,4,9,10,15,16). This is done by using the function isRedundant(c, C), which checks whether the linear constraint $c \in \mathbb{C}_\mathbb{D}$ is redundant with respect to the set C by testing $\alpha_{\mathbb{C}_\mathbb{D}}(C) \sqsubseteq_\mathbb{D} \alpha_{\mathbb{C}_\mathbb{D}}(\{c\})$. Note that the tree unification does not lose any information.

▶ **Example 2.** After tree unification of t_1 and t_2 from Example 1, we obtain:

$$t_1 = [\![\mathtt{A} \geq 4 : \ll[y \geq 2]\gg, [\![\mathtt{A} \geq 2 : \ll[y = 0]\gg, \ll[y = 0]\gg]\!]\!],$$
$$t_2 = [\![\mathtt{A} \geq 4 : \ll[y \geq 0]\gg, [\![\mathtt{A} \geq 2 : \ll[y \geq 0]\gg, \ll[y \leq 0]\gg]\!]\!]$$

Note that the tree unification adds a decision node for $\mathtt{A} \geq 2$ to the right subtree of t_1, whereas it adds a decision node for $\mathtt{A} \geq 4$ to t_2 and removes the redundant constraint $\mathtt{A} \geq 2$ from the resulting left subtree of t_2. ⌟

Some binary operations are performed leaf-wise on the unified decision trees. Given two unified decision trees t_1 and t_2, their ordering $t_1 \sqsubseteq_\mathbb{T} t_2$, join $t_1 \sqcup_\mathbb{T} t_2$, and meet $t_1 \sqcap_\mathbb{T} t_2$ are defined recursively:

$$\ll d_1 \gg \sqsubseteq_\mathbb{T} \ll d_2 \gg = d_1 \sqsubseteq_\mathbb{D} d_2, \qquad [\![c : tl_1, tr_1]\!] \sqsubseteq_\mathbb{T} [\![c : tl_2, tr_2]\!] = (tl_1 \sqsubseteq_\mathbb{T} tl_2) \wedge (tr_1 \sqsubseteq_\mathbb{T} tr_2)$$
$$\ll d_1 \gg \sqcup_\mathbb{T} \ll d_2 \gg = \ll d_1 \sqcup_\mathbb{D} d_2 \gg, \qquad [\![c : tl_1, tr_1]\!] \sqcup_\mathbb{T} [\![c : tl_2, tr_2]\!] = [\![c : tl_1 \sqcup_\mathbb{T} tl_2, tr_1 \sqcup_\mathbb{T} tr_2]\!]$$
$$\ll d_1 \gg \sqcap_\mathbb{T} \ll d_2 \gg = \ll d_1 \sqcap_\mathbb{D} d_2 \gg, \qquad [\![c : tl_1, tr_1]\!] \sqcap_\mathbb{T} [\![c : tl_2, tr_2]\!] = [\![c : tl_1 \sqcap_\mathbb{T} tl_2, tr_1 \sqcap_\mathbb{T} tr_2]\!]$$

The top is a tree with a single $\top_\mathbb{D}$ leaf: $\top_\mathbb{T} = \ll \top_\mathbb{D} \gg$, while the bottom is a tree with a single $\bot_\mathbb{D}$ leaf: $\bot_\mathbb{T} = \ll \bot_\mathbb{D} \gg$.

▶ **Example 3.** Consider the unified trees t_1 and t_2 from Example 2. We have that $t_1 \sqsubseteq_{\mathbb{T}} t_2$ holds, $t_1 \sqcup_{\mathbb{T}} t_2 = [\![\mathtt{A} \geq 4 : \ll [y \geq 0] \gg, [\![\mathtt{A} \geq 2 : \ll [y \geq 0] \gg, \ll [y \leq 0] \gg]\!]]\!]$, and $t_1 \sqcap_{\mathbb{T}} t_2 = [\![\mathtt{A} \geq 4 : \ll [y \geq 2] \gg, [\![\mathtt{A} \geq 2 : \ll [y = 0] \gg, \ll [y = 0] \gg]\!]]\!]$. ⌟

The concretization function $\gamma_{\mathbb{T}}$ is monotonic with respect to the ordering $\sqsubseteq_{\mathbb{T}}$.

▶ **Lemma 4.** $\forall t_1, t_2 \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D})$*:* $t_1 \sqsubseteq_{\mathbb{T}} t_2 \implies \gamma_{\mathbb{T}}(t_1) \subseteq \gamma_{\mathbb{T}}(t_2)$.

Proof. Let $t_1, t_2 \in \mathbb{T}$ such that $t_1 \sqsubseteq_{\mathbb{T}} t_2$. The ordering $\sqsubseteq_{\mathbb{T}}$ between decision trees is implemented by first calling the tree unification algorithm, and then by comparing the decision trees "leaf-wise". Tree unification forces the same structure on decision trees, so all paths to the leaf nodes coincide between the unified decision trees. Let $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ denote the set of linear constraints satisfied along a path of the unified decision trees, and let $d_1, d_2 \in \mathbb{D}_{Var \cup \mathbb{F}}$ denote the leaf nodes reached following the path C within the first and the second decision tree. Since $t_1 \sqsubseteq_{\mathbb{T}} t_2$, we have that $d_1 \sqsubseteq_{\mathbb{D}} d_2$ and so $\gamma_{\mathbb{D}}(d_1) \subseteq \gamma_{\mathbb{D}}(d_2)$. The proof follows from: $\{\langle \sigma, k \rangle \mid \langle \sigma, k \rangle \in \gamma_{\mathbb{D}}(d_1), k \models C\} \subseteq \{\langle \sigma, k \rangle \mid \langle \sigma, k \rangle \in \gamma_{\mathbb{D}}(d_2), k \models C\}$. ◀

Basic Transfer functions

We define basic lifted transfer functions for forward assignments ($\text{ASSIGN}_{\mathbb{T}}$) and tests ($\text{FILTER}_{\mathbb{T}}$), when only program variables occur in given assignments and tests (boolean expressions). Those basic transfer functions $\text{ASSIGN}_{\mathbb{T}}$ and $\text{FILTER}_{\mathbb{T}}$ modify only leaf nodes since the analysis information about program variables is located in leaf nodes while the information about features is located in both decision nodes and leaf nodes.

Algorithm 2 $\texttt{ASSIGN}_{\mathbb{T}}(t, \mathtt{x} := ae, C)$ **when** $\mathbf{vars}(ae) \subseteq \boldsymbol{Var}$.

```
1 if isLeaf(t) then return ≪ASSIGN_{D_{Var∪F}}(t, x:=ae)≫;
2 if isNode(t) then
3     l = ASSIGN_T(t.l, x:=ae, C ∪ {t.c});
4     r = ASSIGN_T(t.r, x:=ae, C ∪ {¬t.c});
5     return [[t.c : l, r]]
```

Algorithm 3 $\texttt{FILTER}_{\mathbb{T}}(t, be, C)$ **when** $\mathbf{vars}(be) \subseteq \boldsymbol{Var}$.

```
1 if isLeaf(t) then return ≪FILTER_{D_{Var∪F}}(t, be)≫;
2 if isNode(t) then
3     l = FILTER_T(t.l, be, C ∪ {t.c});
4     r = FILTER_T(t.r, be, C ∪ {¬t.c});
5     return [[t.c : l, r]]
```

Basic transfer function $\text{ASSIGN}_{\mathbb{T}}$ for handling an assignment $\mathtt{x} := ae$ is described by Algorithm 2. Note that $\mathtt{x} \in \mathit{Var}$ is a program variable, and $ae \in \mathit{AExp}$ may contain only program variables, i.e. the set of variables that occur in ae is $\text{vars}(ae) \subseteq \mathit{Var}$. $\text{ASSIGN}_{\mathbb{T}}$ descends along the paths of the decision tree t up to a leaf node d, where $\text{ASSIGN}_{\mathbb{D}_{Var \cup \mathbb{F}}}$ is invoked to substitute expression ae for variable $\mathtt{x}$ in d. Similarly, basic transfer function $\text{FILTER}_{\mathbb{T}}$ for handling tests $be \in \mathit{BExp}$ when $\text{vars}(be) \subseteq \mathit{Var}$, given in Algorithm 3, is implemented by applying $\text{FILTER}_{\mathbb{D}_{Var \cup \mathbb{F}}}$ leaf-wise, so that be is satisfied by all leaves.

Algorithm 4 $\texttt{FILTER}_{\mathbb{T}}(t, be, C)$ **when** $\textbf{vars}(be) \subseteq \mathbb{F}$.

```
1 switch be do
2     case (ae_0 ⋈ ae_1) || (¬(ae_0 ⋈ ae_1)) do
3         J = FILTER_{D_F}(⊤_{D_F}, be); return RESTRICT(t, C, J)
4     case be_1 ∧ be_2 do
5         return FILTER_T(t, be_1, C) ⊓_T FILTER_T(t, be_2, C)
6     case be_1 ∨ be_2 do
7         return FILTER_T(t, be_1, C) ⊔_T FILTER_T(t, be_2, C)
```

Note that, in program families with static feature binding, features occur only in presence conditions (tests) of `#if` directives. Thus, special transfer functions FEAT-FILTER$_{\mathbb{T}}$ for feature-based tests and IFDEF$_{\mathbb{T}}$ for `#if` directives are defined in [21], which can add, modify, or delete decision nodes of a decision tree. Therefore, the basic transfer function FILTER$_{\mathbb{T}}$ for handling tests $be \in BExp$ when $\text{vars}(be) \subseteq \mathbb{F}$ coincides with FEAT-FILTER$_{\mathbb{T}}$ in [21], and is given in Algorithm 4. It reasons by induction on the structure of be. When be is a comparison of arithmetic expressions (Lines 2,3), we use FILTER$_{\mathbb{D}_{\mathbb{F}}}$ to approximate be, thus producing a set of constraints J, which are then added to the tree t, possibly discarding all paths of t that do not satisfy be. This is done by calling function $\texttt{RESTRICT}(t, C, J)$, which adds linear constraints from J to t in ascending order with respect to $<_{\mathbb{C}_{\mathbb{D}}}$ as shown in Algorithm 5. Note that be may not be representable exactly in $\mathbb{C}_{\mathbb{D}}$ (e.g., in the case of non-linear constraints over $\mathbb{F}$), so FILTER$_{\mathbb{D}_{\mathbb{F}}}$ may produce a set of constraints approximating it. When be is a conjunction (resp., disjunction) of two feature expressions (Lines 4,5) (resp., (Lines 6,7)), the resulting decision trees are merged by operation meet $\sqcap_{\mathbb{T}}$ (resp., join $\sqcup_{\mathbb{T}}$).

The above transfer function and some of the remaining operations rely on function `RESTRICT` given in Algorithm 5 for constraining a decision tree t with respect to a given set J of linear constraints over $\mathbb{F}$. The subtrees whose paths from the root satisfy these constraints are preserved, while leafs of the other subtrees are replaced with bottom $\bot_{\mathbb{D}}$. Function $\texttt{RESTRICT}(t, C, J)$ takes as input a decision tree t, a set C of constraints accumulated along paths up to a node, a set J of linear constraints in canonical form that need to be added to t. For each constraint $j \in J$, there exists a boolean b_j that shows whether the tree should be constrained with respect to j (b_j is set to true) or with respect to $\neg j$ (b_j is set to false). At each iteration, the smallest linear constraint j is extracted from J (Line 9), and is handled appropriately based on whether j is smaller or equal (Line 11–15), or greater (Line 17–21) to the constraint at the node of t we currently consider.

4.2 Extended transfer functions

We now define extended transfer functions ASSIGN$_{\mathbb{T}}$ and FILTER$_{\mathbb{T}}$ where assignments and tests may contain both feature and program variables.

Assignments

Transfer function $\text{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, C)$, when $\text{vars}(ae) \subseteq Var \cup \mathbb{F}$, is given in Algorithm 6. It accumulates the constraints along the paths in the decision tree t in a set of constraints $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ (Lines 8–10), which is initialized to $\mathbb{K}$, up to the leaf nodes in which assignment is performed by $\text{ASSIGN}_{\mathbb{D}_{Var \cup \mathbb{F}}}$. That is, we first merge constraints from the leaf node t

Algorithm 5 $\texttt{RESTRICT}(t, C, J)$.

```
1  if isEmpty(J) then
2      if isLeaf(t) then return t;
3      if isRedundant(t.c, C) then return RESTRICT(t.l, C, J);
4      if isRedundant(¬t.c, C) then return RESTRICT(t.r, C, J);
5      l = RESTRICT(t.l, C ∪ {t.c}, J) ;
6      r = RESTRICT(t.r, C ∪ {¬t.c}, J) ;
7      return ([[t.c : l, r]]);
8  else
9      j = min_{<_{C_D}}(J) ;
10     if isLeaf(t) ∨ (isNode(t) ∧ j ≤_{C_D} t.c) then
11         if isRedundant(j, C) then return RESTRICT(t, C, J\{j});
12         if isRedundant(¬j, C) then return ≪⊥_A≫;
13         if j =_{C_D} t.c then (if b_j then t = t.l else t = t.r) ;
14         if b_j then return ([[j : RESTRICT(t, C ∪ {j}, J\{j}), ≪⊥_A≫]]) ;
15         else return ([[j :≪⊥_A≫, RESTRICT(t, C ∪ {¬j}, J\{j})]]) ;
16     else
17         if isRedundant(t.c, C) then return RESTRICT(t.l, C, J);
18         if isRedundant(¬t.c, C) then return RESTRICT(t.r, C, J);
19         l = RESTRICT(t.l, C ∪ {t.c}, J) ;
20         r = RESTRICT(t.r, C ∪ {¬t.c}, J) ;
21         return ([[t.c : l, r]]);
```

defined over $Var \cup \mathbb{F}$ and constraints from decision nodes $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}_\mathbb{F}})$ defined over $\mathbb{F}$, by using $\uplus_{Var\cup\mathbb{F}}$ operator. Thus, we obtain an abstract element from $\mathbb{D}_{Var\cup\mathbb{F}}$ on which the assignment operator of the domain $\mathbb{D}_{Var\cup\mathbb{F}}$ is applied (Line 2).

Transfer function $\text{ASSIGN}_\mathbb{T}(t, \texttt{A:=}ae, C)$, when $\text{vars}(ae) \subseteq Var \cup \mathbb{F}$, is implemented by Algorithm 7. It calls the auxiliary function $\text{ASSIGN-AUX}_\mathbb{T}(t, \texttt{A:=}ae, C)$, which performs the assignment on each leaf node t merged with the set of linear constraints C collected along the path to the leaf (Line 6). The obtained result d' is a new leaf node (Line 7), and furthermore it is projected on feature variables using $\restriction_\mathbb{F}$ operator to generate a new set of constraints $J = \gamma_{\mathbb{C}_\mathbb{D}}(d' \restriction_\mathbb{F})$ that needs to be substituted to C in the decision tree (Lines 8–13). The substitution is done at each decision node, such that new sets of constraints J_1 and J_2 are collected from its left and right subtrees, and then they are used as constraints in the given decision node instead of $t.c$ and $\neg t.c$. Let $J = J_1 \cap J_2$ be the common (overlapping) set of constraints that arise due to non-determinism (Line 11). When both $J_1 \backslash J$ and $J_2 \backslash J$ are empty, the left and the right subtrees are joined (Line 12). Otherwise, the corresponding tree is constructed using sets $J_1 \backslash J$ and $J_2 \backslash J$ and together with the set J are propagated to the parent node (Line 13). Note that, if some of the sets of constraints J, $J_1 \backslash J$, and $J_2 \backslash J$ is empty in the returned trees in Lines 12-13, then it is considered as a true constraint so that its true branch is always taken.

Tests

Transfer function $\text{FILTER}_\mathbb{T}(t, be, C)$, when $\text{vars}(be) \subseteq Var \cup \mathbb{F}$, is described by Algorithm 8. Similarly to $\text{ASSIGN}_\mathbb{T}(t, \texttt{x:=}ae, C)$ in Algorithm 6, it accumulates the constraints along the paths in the decision tree t in a set of constraints $C \in \mathcal{P}(\mathbb{C}_\mathbb{D})$ up to the leaf nodes (Lines

Algorithm 6 $\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, C)$ **when** $\textbf{vars}(ae) \subseteq \boldsymbol{Var} \cup \mathbb{F}$.

1 **if** isLeaf(t) **then**
2 $d' = \texttt{ASSIGN}_{\mathbb{D}_{Var \cup \mathbb{F}}}(t \uplus_{Var \cup \mathbb{F}} \alpha_{\mathbb{C}_{\mathbb{D}}}(C), \texttt{x:=}ae)$;
3 **return** $\ll d' \gg$
4 **if** isNode(t) **then**
5 $l = \text{ASSIGN}_{\mathbb{T}}(t.l, \texttt{x:=}ae, C \cup \{t.c\})$;
6 $r = \text{ASSIGN}_{\mathbb{T}}(t.r, \texttt{x:=}ae, C \cup \{\neg t.c\})$;
7 **return** $[\![t.c : l, r]\!]$

Algorithm 7 $\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{A:=}ae, C)$ **when** $\textbf{vars}(ae) \subseteq \boldsymbol{Var} \cup \mathbb{F}$.

1 (t,d) = $\texttt{ASSIGN-AUX}_{\mathbb{T}}(t, \texttt{A:=}ae, C)$
2 **return** t
3
4 **Function** $\texttt{ASSIGN-AUX}_{\mathbb{T}}(t, \texttt{A:=}ae, C)$**:**
5 **if** isLeaf(t) **then**
6 $d' = \texttt{ASSIGN}_{\mathbb{D}_{\boldsymbol{Var} \cup \mathbb{F}}}(t \uplus_{\boldsymbol{Var} \cup \mathbb{F}} \alpha_{\mathbb{C}_{\mathbb{D}}}(C), \texttt{A:=}ae)$
7 **return** $(\ll d' \gg, \gamma_{\mathbb{C}_{\mathbb{D}}}(d' \restriction_{\mathbb{F}}))$
8 **if** isNode(t) **then**
9 $(t_1, J_1) = \texttt{ASSIGN-AUX}_{\mathbb{T}}(t.l, \texttt{A:=}ae, C \cup \{t.c\})$
10 $(t_2, J_2) = \texttt{ASSIGN-AUX}_{\mathbb{T}}(t.r, \texttt{A:=}ae, C \cup \{\neg t.c\})$
11 $J = J_1 \cap J_2$
12 **if** isEmpty($J_1 \backslash J$) $\wedge$ isEmpty($J_2 \backslash J$) **then return** $\left([\![J, t_1 \sqcup_{\mathbb{T}} t_2, \bot_{\mathbb{T}}]\!], \emptyset\right)$
13 **else return** $\left([\![J_1 \backslash J, t_1, [\![J_2 \backslash J, t_2, \bot_{\mathbb{T}}]\!]]\!], J\right)$

6–9). When t is a leaf node, test be is handled using $\text{FILTER}_{\mathbb{D}_{Var \cup \mathbb{F}}}$ applied on an abstract element from $\mathbb{D}_{Var \cup \mathbb{F}}$ obtained by merging constraints in the leaf node and decision nodes along the path to the leaf (Lines 2). The obtained result d' represents a new leaf node, and furthermore d' is projected on feature variables using $\restriction_{\mathbb{F}}$ operator to generate a new set of constraints J that is added to the given path to d' (Lines 3–5).

Note that the trees returned by $\text{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, C)$, $\text{ASSIGN}_{\mathbb{T}}(t, \texttt{A:=}ae, C)$, and $\text{FILTER}_{\mathbb{T}}(t, be, C)$ are sorted (normalized) to remove possible multiple occurrences of a constraint c, possible occurrences of both c and $\neg c$, and possible ordering inconsistences. Moreover, the obtained decision trees may contain some redundancy that can be exploited to further compress them. We use several optimizations [21, 45]. E.g., if constraints on a path to some leaf are unsatisfiable, we eliminate that leaf node; if a decision node contains two same subtrees, then we keep only one subtree and we also eliminate the decision node, etc.

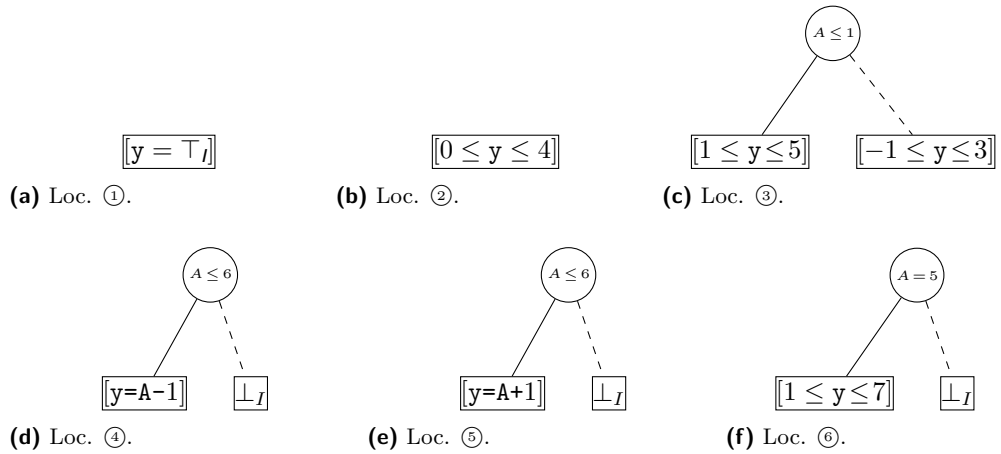
▶ **Example 5.** Let us consider the following dynamic program family P':

```
①  int y := [0,4];
②  if (A < 2) y := y+1; else y := y-1;
③  A := y+1;
④  y := A+1;
⑤  A := 5;   ⑥
```

The code base of P' contains only one program variable $Var = \{\texttt{y}\}$ and one numerical feature $\mathbb{F} = \{A\}$ with domain $dom(A) = [0, 99]$. In Fig. 5 we depict decision trees inferred by performing *polyhedral* lifted analysis using the lifted domain $\mathbb{T}(\mathbb{C}_P, P)$. We use $\text{FILTER}_{\mathbb{T}}$

Algorithm 8 $\texttt{FILTER}_{\mathbb{T}}(t, be, C)$ **when** $\textbf{vars}(be) \subseteq \boldsymbol{Var} \cup \mathbb{F}$.

```
1 if isLeaf(t) then
2     d' = FILTER_{D_{Var∪F}}(t ⊎_{Var∪F} α_{C_D}(C), be);
3     J = γ_{C_D}(d' ↾_F);
4     if isRedundant(J, C) then return ≪d'≫;
5     else return RESTRICT(≪d'≫, C, J\C);
6 if isNode(t) then
7     l = FILTER_T(t.l, be, C ∪ {t.c});
8     r = FILTER_T(t.r, be, C ∪ {¬t.c});
9     return [[t.c : l, r]]
```



(a) Loc. ①. **(b)** Loc. ②. **(c)** Loc. ③.

(d) Loc. ④. **(e)** Loc. ⑤. **(f)** Loc. ⑥.

Figure 5 Decision tree-based (polyhedral) invariants at program locations from ① to ⑥ of P'.

from Algorithm 4 to analyze statement at location ② and infer the decision tree at location ③. Then, we use $\texttt{ASSIGN}_{\mathbb{T}}$ from Algorithm 7 to analyze the statement A `:= y+1` at ③ and infer the tree at location ④. Note that, by using the left and right leafs in the input tree at ③, we generate constraint sets $J_1 = (2 \leq A \leq 6)$ and $J_2 = (0 \leq A \leq 4)$ with the same leaf nodes [`y=`A`-1`]. After applying reductions, we obtain the tree at location ④. Recall that we implicitly assume the correctness of linear constraints $\mathbb{K}$ that take into account domains of features. Hence, node $(A \leq 6)$ is satisfied when $(A \leq 6) \wedge (0 \leq A \leq 99)$, where constraint $(0 \leq A \leq 99)$ represents the domains of A. Finally, statement `y :=` A`+1` at location ④ is analyzed using Algorithm 6 such that all leafs in the input tree are updated accordingly, whereas statement A `:=` 5 at location ⑤ is analyzed using Algorithm 7 such that all leafs in the input tree along the paths to them are joined to create new leaf that satisfies $(A = 5)$.

4.3 Widening

The widening operator $\nabla_{\mathbb{T}}$ is necessary in order to extrapolate an analysis property over configurations (values of features) and stores (values of program variables) on which it is not yet defined. Hence, it provides a way to handle (potentially) infinite reconfiguration of features inside loops. The widening $t_1 \nabla_{\mathbb{T}} t_2$ is implemented by calling function $\texttt{WIDEN}_{\mathbb{T}}(t_1, t_2, \mathbb{K})$, where t_1 and t_2 are two decision trees and $\mathbb{K}$ is the set of valid configurations. Function $\texttt{WIDEN}_{\mathbb{T}}$, given in Algorithm 9, first calls function `LEFT_UNIFICATION` (Line 1) that performs widening of the configuration space (i.e., decision nodes), and then extrapolates the value

of leafs by calling function `WIDEN_LEAF` (Line 2). Function `LEFT_UNIFICATION` (Lines 4–17) limits the size of decision trees, and thus avoids infinite sequences of partition refinements. It forces the structure of t_1 on t_2. This way, there may be information loss by applying this function. `LEFT_UNIFICATION` accumulates into a set C (initially equal to $\mathbb{K}$) the linear constraints along the paths in the first decision tree, possibly adding nodes to the second tree (Lines 10–17), or removing decision nodes from the second tree in which case the left and the right subtree are joined (Lines 6–9), or removing constraints that are redundant (Lines 7,8 and 11,12). Finally, function `WIDEN_LEAF` (Line 18–23) applies the widening $\nabla_{\mathbb{D}}$ leaf-wise on the left unified decision trees.

Algorithm 9 $\texttt{WIDEN}_{\mathbb{T}}(t_1, t_2, C)$.

```
1  (t1, t2) = LEFT_UNIFICATION(t1, t2, C)
2  return WIDEN_LEAF(t1, t2, C)
3
4  Function LEFT_UNIFICATION(t1, t2, C):
5      if isLeaf(t1) ∧ isLeaf(t2) then return (t1, t2)
6      if isLeaf(t1) ∨ (isNode(t1) ∧ isNode(t2) ∧ t2.c <_{C_D} t1.c) then
7          if isRedundant(t2.c, C) then return LEFT_UNIFICATION(t1, t2.l, C)
8          if isRedundant(¬t2.c, C) then return LEFT_UNIFICATION(t1, t2.r, C)
9          return LEFT_UNIFICATION(t1, t2.l ⊔_T t2.r, C)
10     if isLeaf(t2) ∨ (isNode(t1) ∧ isNode(t2) ∧ t1.c ≤_{C_D} t2.c) then
11         if isRedundant(t1.c, C) then return UNIFICATION(t1.l, t2, C)
12         if isRedundant(¬t1.c, C) then return UNIFICATION(t1.r, t2, C)
13         if t1.c <_{C_D} t2.c then  t21 = t2; t22 = t2;
14         else t21 = t2.l; t22 = t2.r;
15         (l1, l2) = UNIFICATION(t1.l, t21, C ∪ {t1.c})
16         (r1, r2) = UNIFICATION(t1.r, t22, C ∪ {¬t1.c})
17         return ([[t1.c : l1, r1]], [[t1.c : l2, r2]])
18 Function WIDEN_LEAF(t1, t2, C):
19     if isLeaf(t1) ∧ isLeaf(t2) then return (≪t1 ∇_D t2≫)
20     if isNode(t1) ∧ isNode(t2) then
21         l = WIDEN_LEAF(t1.l, t2.l, C ∪ {t1.c})
22         r = WIDEN_LEAF(t1.r, t2.r, C ∪ {¬t1.c})
23         return ([[t1.c : l, r]])
```

▶ **Example 6.** Consider the following two decision trees t_1 and t_2:

$$t_1 = [\![\mathtt{A}>1 : [\![\mathtt{A}>5 : \ll[y\geq 0]\gg, \ll[y\leq 0]\gg]\!], \ll[y=0]\gg]\!]$$
$$t_2 = [\![\mathtt{A}>2 : \ll[y=1]\gg, \ll[y>1]\gg]\!]$$

After applying the left unification of t_1 and t_2, the tree t_2 becomes:

$$t_2 = [\![\mathtt{A}>1 : [\![\mathtt{A}>5 : \ll[y=1]\gg, \ll[y\geq 1]\gg]\!], \ll[y>1]\gg]\!]$$

Note that when $(\mathtt{A}>1)$ and $\neg(\mathtt{A}>5)$, the left and right leafs of the input t_2 are joined, thus yielding the leaf $[y\geq 1]$ in the left-unified t_2. This represents an example of information-loss in a left-unified tree. After applying the leaf-wise widening of t_1 and left-unified t_2, we obtain:

$$t = [\![\mathtt{A}>1 : [\![\mathtt{A}>5 : \ll[y\geq 0]\gg, \ll\top\gg]\!], \ll[y\geq 0]\gg]\!]$$ ⌟

$$
\begin{aligned}
[\![\texttt{skip}]\!]^\sharp t &= t \\
[\![\texttt{x := } ae]\!]^\sharp t &= \texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, \mathbb{K}) \\
[\![s_1 \texttt{ ; } s_2]\!]^\sharp t &= [\![s_2]\!]^\sharp([\![s_1]\!]^\sharp t) \\
[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]^\sharp t &= [\![s_1]\!]^\sharp(\texttt{FILTER}_{\mathbb{T}}(t, be, \mathbb{K})) \sqcup_{\mathbb{T}} [\![s_2]\!]^\sharp(\texttt{FILTER}_{\mathbb{T}}(t, \neg be, \mathbb{K})) \\
[\![\texttt{while } be \texttt{ do } s]\!]^\sharp t &= \texttt{FILTER}_{\mathbb{T}}(\mathrm{lfp}^\sharp\, \phi^\sharp, \neg be, \mathbb{K}) \\
\phi^\sharp(x) &= t \sqcup_{\mathbb{T}} [\![s]\!]^\sharp(\texttt{FILTER}_{\mathbb{T}}(x, be, \mathbb{K})) \\
[\![A \texttt{ := } ae]\!]^\sharp t &= \texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{A:=}ae, \mathbb{K})
\end{aligned}
$$

Figure 6 Abstract invariance semantics $[\![s]\!]^\sharp : \mathbb{T} \to \mathbb{T}$.

4.4 Soundness

The operations and transfer functions of the decision tree lifted domain $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D})$ can now be used to define the abstract invariance semantics. In Fig. 6, we define the *abstract invariance semantics* $[\![s]\!]^\sharp : \mathbb{T} \to \mathbb{T}$ for each statement s. Function $[\![s]\!]^\sharp$ takes as input a decision tree over-approximating the set of reachable states at the initial location of statement s, and outputs a decision tree that over-approximates the set of reachable states at the final location od s. For a `while` loop, $\mathrm{lfp}^\sharp\, \phi^\sharp$ is the limit of the following increasing chain defined by the widening operator $\nabla_{\mathbb{T}}$ (note that, $t_1 \nabla_{\mathbb{T}} t_2 = \texttt{WIDEN}_{\mathbb{T}}(t_1, t_2, \mathbb{K})$):

$$y_0 = \bot_{\mathbb{T}}, \quad y_{n+1} = y_n \nabla_{\mathbb{T}} \phi^\sharp(y_n)$$

The lifted analysis (abstract invariance semantics) of a dynamic program family s is defined as $[\![s]\!]^\sharp t_{in}$, where the input tree t_{in} at the initial location has only one leaf node $\top_{\mathbb{D}}$ and decision nodes define the set $\mathbb{K}$. Note that $t_{in} = \ll\top_{\mathbb{D}}\gg$ if there are no constraints in $\mathbb{K}$. This way, by calculating $[\![s]\!]^\sharp t_{in}$ we collect the possible invariants in the form of decision trees at all program locations.

We can establish soundness of the abstract invariant semantics $[\![s]\!]^\sharp t_{in} \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D})$ with respect to the invariance semantics $[\![s]\!]\langle\Sigma, \mathbb{K}\rangle \in \mathcal{P}(\Sigma \times \mathbb{K})$, where $\langle\Sigma, \mathbb{K}\rangle = \{\langle\sigma, k\rangle \mid \sigma \in \Sigma, k \in \mathbb{K}\}$, by showing that $[\![s]\!]\langle\Sigma, \mathbb{K}\rangle \subseteq \gamma_{\mathbb{T}}\big([\![s]\!]^\sharp t_{in}\big)$. This is done by proving the following result. [2]

▶ **Theorem 7** (Soundness). $\forall t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D}) : [\![s]\!]\gamma_{\mathbb{T}}(t) \subseteq \gamma_{\mathbb{T}}\big([\![s]\!]^\sharp t\big)$.

Proof. The proof is by structural induction on s. We consider the most interesting cases.

Case `skip`. $[\![\texttt{skip}]\!]\gamma_{\mathbb{T}}(t) = \gamma_{\mathbb{T}}(t) = \gamma_{\mathbb{T}}([\![\texttt{skip}]\!]^\sharp t)$.

Case `x:=`ae. Let $\langle\sigma, k\rangle \in \gamma_{\mathbb{T}}(t)$. By definition of $[\![\texttt{x := } ae]\!]$ in Fig. 4, it holds that $\langle\sigma[\texttt{x} \mapsto n], k\rangle \in [\![\texttt{x := } ae]\!]\gamma_{\mathbb{T}}(t)$ for all $n \in [\![ae]\!]\langle\sigma, k\rangle$. Since $\langle\sigma, k\rangle \in \gamma_{\mathbb{T}}(t)$, there must be a leaf node d of t and a set of constraints C collected along the path to d, such that $\langle\sigma, k\rangle \in \gamma_{\mathbb{D}}(d) \wedge k \models C$. By definition of the abstraction $\langle\mathcal{P}(\Sigma \times \mathbb{K}), \subseteq\rangle \xleftarrow{\gamma_{\mathbb{D}}} \langle\mathbb{D}_{Var\cup\mathbb{F}}, \sqsubseteq_{\mathbb{D}}\rangle$, the soundness of $\texttt{ASSIGN}_{\mathbb{D}_{Var\cup\mathbb{F}}}$, and by definition of $\texttt{ASSIGN}_{\mathbb{T}}$ (cf. Algorithms 2 and 6), it must hold $\langle\sigma[\texttt{x} \mapsto n], k\rangle \in \gamma_{\mathbb{T}}(\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x := } ae, \mathbb{K}))$ due to the fact that Algorithms 2 and 6 invoke $\texttt{ASSIGN}_{\mathbb{D}_{Var\cup\mathbb{F}}}$ for every leaf node of t that may be merged with linear constraints from decision nodes found on the path from the root to that leaf. Thus, we conclude $[\![\texttt{x := } ae]\!]\gamma_{\mathbb{T}}(t) \subseteq \gamma_{\mathbb{T}}(\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, \mathbb{K})) = \gamma_{\mathbb{T}}([\![\texttt{x := } ae]\!]^\sharp t)$.

[2] Note that $\gamma_{\mathbb{T}}(t_{in}) = \langle\Sigma, \mathbb{K}\rangle$.

Case `if` be `then` s_1 `else` s_2. Let $\langle\sigma,k\rangle \in \gamma_{\mathbb{T}}(t)$ and $\langle\sigma',k'\rangle \in [\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]\{\langle\sigma,k\rangle\}$. By structural induction, we have that $[\![s_1]\!]\gamma_{\mathbb{T}}(t') \subseteq \gamma_{\mathbb{T}}([\![s_1]\!]^\sharp t')$ and $[\![s_2]\!]\gamma_{\mathbb{T}}(t') \subseteq \gamma_{\mathbb{T}}([\![s_2]\!]^\sharp t')$ for any t'. By definition of $[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]$ in Fig. 4, we have that $\langle\sigma',k'\rangle \in [\![s_1]\!]\{\langle\sigma,k\rangle\}$ if true $\in [\![be]\!]\langle\sigma,k\rangle$ or $\langle\sigma',k'\rangle \in [\![s_2]\!]\{\langle\sigma,k\rangle\}$ if false $\in [\![be]\!]\langle\sigma,k\rangle$. Since $\langle\sigma,k\rangle \in \gamma_{\mathbb{T}}(t)$, there must be a leaf node d of t and a set of constraints C collected along the path to d, such that $\langle\sigma,k\rangle \in \gamma_{\mathbb{D}}(d) \wedge k \models C$. By definition of the abstraction $\langle\mathcal{P}(\Sigma\times\mathbb{K}),\subseteq\rangle \xleftarrow{\gamma_{\mathbb{D}}} \langle\mathbb{D}_{Var\cup\mathbb{F}},\sqsubseteq_{\mathbb{D}}\rangle$, the soundness of $\texttt{FILTER}_{\mathbb{D}_{Var\cup\mathbb{F}}}$, and by definition of $\texttt{FILTER}_{\mathbb{T}}$ (cf. Algorithms 2, 4, and 8), it must hold that $\langle\sigma,k\rangle \in \gamma_{\mathbb{T}}(\texttt{FILTER}_{\mathbb{T}}(t,be,\mathbb{K}))$ or $\langle\sigma,k\rangle \in \gamma_{\mathbb{T}}(\texttt{FILTER}_{\mathbb{T}}(t,\neg be,\mathbb{K}))$ due to the fact that these Algorithms invoke $\texttt{FILTER}_{\mathbb{D}_{Var\cup\mathbb{F}}}$ for every leaf node of t that may be merged with linear constraints from decision nodes found on the path from the root to that leaf. Thus, by structural induction, we have $\langle\sigma',k'\rangle \in \gamma_{\mathbb{T}}([\![s_1]\!]^\sharp\texttt{FILTER}_{\mathbb{T}}(t,be,\mathbb{K}))$ or $\langle\sigma',k'\rangle \in \gamma_{\mathbb{T}}([\![s_2]\!]^\sharp\texttt{FILTER}_{\mathbb{T}}(t,\neg be,\mathbb{K}))$, and so $\langle\sigma',k'\rangle \in \gamma_{\mathbb{T}}([\![s_1]\!]^\sharp\texttt{FILTER}_{\mathbb{T}}(t,be,\mathbb{K}) \sqcup_{\mathbb{T}} [\![s_2]\!]^\sharp\texttt{FILTER}_{\mathbb{T}}(t,\neg be,\mathbb{K}))$. Thus, we conclude that $[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]\gamma_{\mathbb{T}}(t) \subseteq \gamma_{\mathbb{T}}([\![s_1]\!]^\sharp\texttt{FILTER}_{\mathbb{T}}(t,be,\mathbb{K}) \sqcup_{\mathbb{T}} [\![s_2]\!]^\sharp\texttt{FILTER}_{\mathbb{T}}(t,\neg be,\mathbb{K})) = \gamma_{\mathbb{T}}([\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]^\sharp t)$.

Case `while` e `do` s. We show that, given a $t \in \mathbb{T}$, for all $x \in \mathbb{T}$, we have: $\phi(\gamma_{\mathbb{T}}(x)) \subseteq \gamma_{\mathbb{T}}(\phi^\sharp(x))$. By structural induction, we have $[\![s]\!]\gamma_{\mathbb{T}}(x) \subseteq \gamma_{\mathbb{T}}([\![s]\!]^\sharp x)$.
Let $\langle\sigma,k\rangle \in \gamma_{\mathbb{T}}(x)$ and $\langle\sigma',k'\rangle \in \phi(\gamma_{\mathbb{T}}(x))$. By definition of $\phi(x)$ in Fig. 4, we have that $\langle\sigma',k'\rangle \in [\![s]\!]\{\langle\sigma,k\rangle\}$ and true $\in [\![be]\!]\langle\sigma,k\rangle$. By definition of the abstraction $\langle\mathcal{P}(\Sigma\times\mathbb{K}),\subseteq\rangle \xleftarrow{\gamma_{\mathbb{D}}} \langle\mathbb{D}_{Var\cup\mathbb{F}},\sqsubseteq_{\mathbb{D}}\rangle$, the soundness of $\texttt{FILTER}_{\mathbb{D}_{Var\cup\mathbb{F}}}$, and by definition of $\texttt{FILTER}_{\mathbb{T}}$ (cf. Algorithms 2, 4, and 8), it must hold that $\langle\sigma,k\rangle \in \gamma_{\mathbb{T}}(\texttt{FILTER}_{\mathbb{T}}(x,be,\mathbb{K}))$ by using similar arguments to "`if`" case. Thus, by structural induction, we have $\langle\sigma',k'\rangle \in \gamma_{\mathbb{T}}([\![s]\!]^\sharp\texttt{FILTER}_{\mathbb{T}}(x,be,\mathbb{K}))$, and so $\langle\sigma',k'\rangle \in \gamma_{\mathbb{T}}(\phi^\sharp(x))$. We conclude $\phi(\gamma_{\mathbb{T}}(x)) \subseteq \gamma_{\mathbb{T}}(\phi^\sharp(x))$. The proof that $[\![\texttt{while } e \texttt{ do } s]\!]\gamma_{\mathbb{T}}(t) \subseteq \gamma_{\mathbb{T}}([\![\texttt{while } e \texttt{ do } s]\!]^\sharp(t))$ follows from the definition of $\nabla_{\mathbb{T}}$ (cf. Algorithm 9) that invokes the sound $\nabla_{\mathbb{D}_{Var\cup\mathbb{F}}}$ operator on leaf nodes. ◀

▶ **Example 8.** Let us consider the following dynamic program family P'':

```
① A := [10, 15];
② int x := 10, y;
③ if (A>12) then y := 1 else y := −1;
④ while ⑤ (x  > 0) {
⑥     A := A+y;
⑦     x := x-1;
⑧ }  ⑨
```

which contains one feature `A` with domain [0,99]. Initially, `A` can have a value from [10,15]. We can calculate the abstract invariant semantics $[\![P'']\!]^\sharp$, thus obtaining invariants from $\mathbb{T}$ in all locations. We show the inferred invariants in locations ⑤ and ⑨ in Figs. 7 and 8, respectively. The decision tree at the final location ⑨ shows that we have $\texttt{x=0} \wedge \texttt{y=1}$ when $23 \leq \texttt{A} \leq 25$ and $\texttt{x=0} \wedge \texttt{y=-1}$ when $0 \leq \texttt{A} \leq 2$ on program exit. On the other hand, if we analyze P'' using single-program polyhedra analysis, where `A` is considered as an ordinary program variable, we obtain the following less precise invariant on program exit: $\texttt{x=0} \wedge -1 \leq \texttt{y} \leq 1 \wedge 5 \leq 2\texttt{A} - 5\texttt{y} \leq 45$. ⌟

5 Evaluation

We evaluate our decision tree-based approach for analyzing dynamic program families by comparing it with the single-program analysis approach, in which dynamic program families

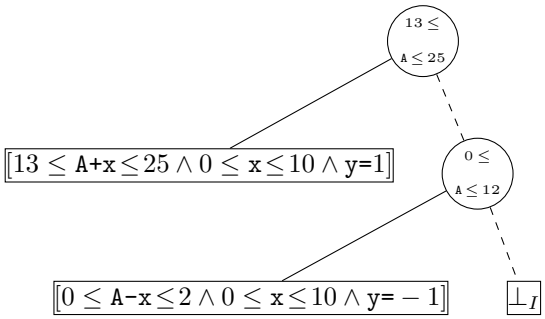


Figure 7 Invariant at loc. ⑤ of P''.

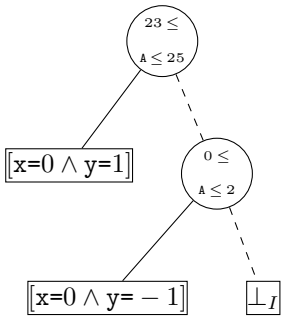


Figure 8 Invariant at loc. ⑨ of P''.

are considered as single programs and features as ordinary program variables. The evaluation aims to show that our decision tree-based approach can effectively analyze dynamic program families and that it achieves a good precision/cost tradeoff with respect to the single-program analysis. Specifically, we ask the following research questions:

RQ1: How precise are inferred invariants of our decision tree-based approach compared to single-program analysis?

RQ2: How time efficient is our decision tree-based approach compared to single-program analysis?

RQ3: Can we find practical application scenarios of using our approach to effectively analyze dynamic program families?

Implementation

We have developed a prototype lifted static analyzer, called DSPLNum²Analyzer, which uses the lifted domain of decision trees $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D})$. The abstract operations and transfer functions of the numerical domain $\mathbb{D}$ (e.g., intervals, octagons, and polyhedra) are provided by the APRON library [33]. Our proof-of-concept implementation is written in OCaml and consists of around 8K lines of code. The current front-end of the tool provides only a limited support for arrays, pointers, recursion, `struct` and `union` types, though an extension is possible. The only basic data type is mathematical integers, which is sufficient for our purposes. DSPLNum²Analyzer automatically computes a decision tree from the lifted domain in every program location. The analysis proceeds by structural induction on the program syntax, iterating `while`-s until a fixed point is reached. We apply delayed widening [13], which means that we start extrapolating by widening only after some fixed number of iterations we explicitly analyze the loop's body. The precision of the obtained invariants for `while`-s is further improved by applying the narrowing operator [13]. We can tune the precision and time efficiency of the analysis by choosing the underlying numerical abstract domain (intervals, octagons, polyhedra), and by adjusting the widening delay. The precision of domains increases from intervals to polyhedra, but so does the computational complexity.

Experimental setup and Benchmarks

All experiments are executed on a 64-bit Intel®Core^{TM} i7-8700 CPU@3.20GHz × 12, Ubuntu 18.04.5 LTS, with 8 GB memory. All times are reported as averages over five independent executions. The implementation, benchmarks, and all results obtained from our experiments are available from [20]: `https://zenodo.org/record/4718697#.YJrDzagzbIU`. We use three instances of our lifted analyses via decision trees: $\overline{\mathcal{A}}_{\mathbb{T}}(I)$, $\overline{\mathcal{A}}_{\mathbb{T}}(O)$, and $\overline{\mathcal{A}}_{\mathbb{T}}(P)$, which use intervals, octagons, and polyhedra domains as parameters. We compare our approach

Table 1 Performance results for single analysis $\mathcal{A}(I)$ vs. lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with one and two features on selected e-mail variant simulators. All times are in seconds.

Benchmark	LOC	$\mathcal{A}(I)$, 0 feature			$\overline{\mathcal{A}}_{\mathbb{T}}(I)$, 1 feature			$\overline{\mathcal{A}}_{\mathbb{T}}(I)$, 2 features		
		Time	Unrea.	Rea.	Time	Unrea.	Mix	Time	Unrea.	Mix
`e-mail_spec0`	2645	16.2	80	48	29.3	80	48(1:1)	50.7	80	48(3:1)
`e-mail_spec6`	2660	18.8	6	26	23.6	16	16(1:1)	24.2	16	16(3:1)
`e-mail_spec8`	2665	14.6	12	20	19.1	12	20(1:1)	27.7	12	20(2:2)
`e-mail_spec11`	2660	15.2	160	96	24.7	160	96(1:1)	32.1	160	96(3:1)
`e-mail_spec27`	2630	14.5	384	128	28.4	384	128(1:1)	38.4	384	128(3:1)

with three instances of the single-program analysis based on numerical domains from the APRON library [33]: $\mathcal{A}(I)$, $\mathcal{A}(O)$, and $\mathcal{A}(P)$, which use intervals, octagons, and polyhedra domains, respectively. The default widening delay is 2.

The evaluation is performed on a dozen of C numerical programs collected from several categories of the 9th International Competition on Software Verification (SV-COMP 2020) [3]: `product lines`, `loops`, `loop-invgen` (`invgen` for short), `loop-lit` (`lit` for short), and `termination-crafted` (`crafted` for short). In the case of `product lines`, we selected the e-mail system [26], which has been used before to assess product-line verification in the product-line community [2, 3, 48]. The e-mail system has eight features: encryption, decryption, automatic forwarding, e-mail signatures, auto responder, keys, verify, and address book, which can be activated or deactivated at run-time. There are forty valid configurations that can be derived. For the other categories, we have first selected some numerical programs, and then we have considered some of their integer variables as features. Basically, we selected those program variables as features that control configuration decisions and can influence the outcome of the given assertions. Tables 1 and 2 present characteristics of the selected benchmarks in our empirical study, such as: the file name (Benchmark), the category where it is located (folder), number of features ($|\mathbb{F}|$), total number of lines of code (LOC).

We use the analyses $\mathcal{A}(\mathbb{D})$ and $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$ to evaluate the validity of assertions in the selected benchmarks. Let $d \in \mathbb{D}$ be a numerical invariant found before the assertion `assert`(be). An analysis can establish that the assertion is: (1) "*unreachable*", if $d = \bot_{\mathbb{D}}$; (2) "*correct*" (valid), if $d \sqsubseteq_{\mathbb{D}} \text{FILTER}_{\mathbb{D}}(d, be)$, meaning that the assertion is indeed valid regardless of approximations; (3) "*erroneous*" (invalid), if $d \sqsubseteq_{\mathbb{D}} \text{FILTER}_{\mathbb{D}}(d, \neg be)$, meaning that the assertion is indeed invalid; and (4) "*I don't know*", otherwise, meaning that the approximations introduced due to abstraction prevent the analyzer from giving a definite answer. We say that an assertion is *reachable* if one of the answers (2), (3), or (4) is obtained. In the case of the lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$, we may also obtain *mixed* assertions when different leaf nodes of the resulting decision trees yield different answers.

Results

E-mail system. We use a *variant simulator* that has been generated with *variability encoding* from the e-mail configurable system [26]. Variability encoding is a process of encoding compile-time (static) variability of a configurable system as run-time (dynamic) variability in the variant simulator [48, 32]. In this setting, compile-time features are encoded with global program variables, and static configuration choices (e.g., `#if`-s) are encoded with

[3] `https://sv-comp.sosy-lab.org/2020/`

conditional statements in the target language (`if` statements). We consider five specifications of the e-mail system encoded as assertions in SV-COMP. As variant simulators use standard language constructs to express variability (`if` statements), they can be analyzed by standard single-program analyzers $\mathcal{A}(\mathbb{D})$. We also analyze the variant simulators using our lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$, where some of the feature variables are considered as real features. This way, our aim is to obtain more precise analysis results. For effectiveness, we consider only those feature variables that influence directly the specification as real features. Specifically, we consider variant simulators with one and two separate features, and five specifications: `spec0`, `spec6`, `spec8`, `spec11`, and `spec27`. For example, `spec0` checks whether a message to be forwarded is readable, while `spec27` checks whether the public key of a person who sent a message is available. For each specification, many assertions appear in the main function after inlining.

Table 1 shows the results of analyzing the selected e-mail simulators using $\mathcal{A}(I)$ and $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with one and two features. In the case of $\mathcal{A}(I)$, we report the number of assertions that are found "unreachable", denoted by Unrea., and reachable ("correct"/"erroneous"/"I don't know"), denoted by Rea.. In the case of $\overline{\mathcal{A}}_{\mathbb{T}}(I)$, we report the number of "unreachable" assertions, denoted by Unrea., and mixed assertions, denoted by Mix. When a reachable ("correct"/"erroneous"/"I don"t know") assertion is reported by $\mathcal{A}(I)$, the lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ may give more precise answer by providing the information for which variants that assertion is reachable and for which is unreachable. We denote by $(n : m)$ the fact that one assertion is unreachable in n variants and reachable in m variants. Note that feature variables in variant simulators are non-deterministically initialized at the beginning of the program and then can be only read in guards of `if` statements, thus $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ may only find more precise answers than $\mathcal{A}(I)$ with respect to the reachability of assertions. That is, it may find more assertions that are unreachable in various variants. See the following paragraph "Other benchmarks" for examples where "I don't know" answers by $\mathcal{A}(I)$ are turned into definite ("correct"/"erroneous") answers by $\overline{\mathcal{A}}_{\mathbb{T}}(I)$. We can see in Table 1 that, for all reachable assertions found by $\mathcal{A}(I)$, we obtain more precise answers using the lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(I)$. For example, $\mathcal{A}(I)$ finds 128 "I don't know" assertions for `spec27`, while $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with one feature `Keys` finds 128 (1:1) mixed assertions such that each assertion is "unreachable" when `Keys`=0 and "I don't know" when `Keys`=1. By using $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with two features `Keys` and `Forward`, we obtain 128 (3:1) mixed assertions, with each assertion is "unreachable" when $\texttt{Keys} = 0 \vee \texttt{Forward} = 0$. Similar analysis results are obtained for the other specifications. For all specifications, the analysis time increases by considering more features. In particular, we find that $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with one feature is in average 1.6 times slower than $\mathcal{A}(I)$, and $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with two features is in average 2.2 times slower than $\mathcal{A}(I)$. However, we also obtain more precise information when using $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with respect to the reachability of assertions in various configurations.

Other benchmarks. We now present the performance results for the benchmarks from other SV-COMP categories. The program `half_2.c` from `loop-invgen` category is given in Fig. 9a. When we perform a single-program analysis $\mathcal{A}(P)$, we obtain the "I don't know" answer for the assertion. However, if n is considered as a feature and the lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ is performed on the resulting dynamic program family, we yield that the assertion is: "correct" when $n \geq 1$, "erroneous" when $n \leq -2$, and "I don't know" answer otherwise. We observe that the lifted analysis considers two different behaviors of `half_2.c` separately: the first when the loops are executed one or more times, and the second when the loops are not executed at all. Hence, we obtain definite answers, "correct" and "erroneous", for the two behaviors. The program `seq.c` from `loop-invgen` category is

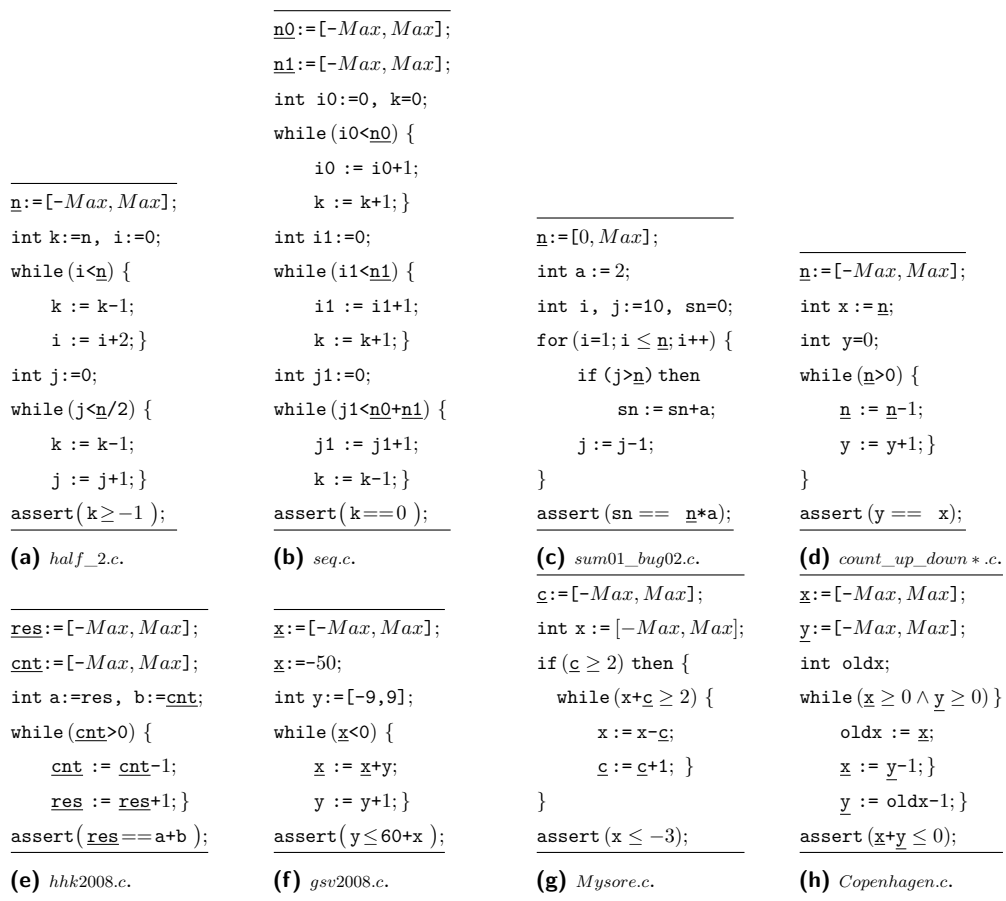


Figure 9 Benchmarks from SV-COMP. All underlined variables are considered as features in the corresponding dynamic program families.

given in Fig. 9b. When `seq.c` is analyzed using $\mathcal{A}(P)$, we obtain "I don't know" for the assertion. When $n0$ and $n1$ are considered as features with the domains $[-Max, +Max]$, $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ gives more precise results for the assertion. In particular, the assertion is "correct" when $(1 \leq n0 \leq Max \wedge 1 \leq n1 \leq Max)$ or $(-Max \leq n0 \leq 0 \wedge -Max \leq n1 \leq 0)$, whereas the assertion is "erroneous" when $(n0 + n1 \leq 0 \wedge (n0 \geq 1 \vee n1 \geq 1))$ and we obtain "I don't know" when $(n0 + n1 \geq 1 \wedge (n0 \leq 0 \vee n1 \leq 0))$. The program `sum01_bug02.c` from `loops` is given in Fig. 9c. $\mathcal{A}(P)$ reports "I don't know" for the assertion, while $\overline{\mathcal{A}}_{\mathbb{T}}(P)$, when n is a feature with domain $[0, Max]$, reports more precise answers: "erroneous" when $n \geq 9$, "correct" when $n = 0$, and "I don't know" otherwise. $\mathcal{A}(P)$ reports "I don't know" for the assertion in `count_up_down*.c` from `loops`, which is given in Fig. 9d. Still, $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ when n is a feature with domain $[-Max, Max]$ reports: "correct" answer when $n = 0$ at the final location, "erroneous" when $n \leq -1$, and "I don't know" otherwise. Similarly, $\mathcal{A}(P)$ reports "I don't know" for the assertions in `hhk2008.c` and `gsv2008.c` from `loop-lit` (given in Figs. 9e and 9f). However, $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ reports more precise answers in both cases. We consider res and cnt (resp., x) as features with domains $[-Max, Max]$ for `hhk2008.c` (resp., `gsv2008.c`), and we obtain "correct" answer when $cnt = 0$ for `hhk2008.c` (resp., when $x \geq 0$ for `gsv2008.c`), "erroneous" answer when $cnt \leq -1$ for `hhk2008.c`, and "I don't know" answer otherwise. Finally, $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ reports more precise answers than $\mathcal{A}(P)$ for `Mysore.c` and `Copenhagen.c` from `termination crafted` category (given in Figs. 9g and 9h).

Table 2 Performance results for single analysis $\mathcal{A}(\mathbb{D})$ vs. lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$ and $\overline{\mathcal{A}}_{\mathbb{T}}(O)$ on selected benchmarks from SV-COMP.All times are in seconds.

Benchmark	folder	$\lvert\mathbb{F}\rvert$	LOC	$\mathcal{A}(P)$		$\overline{\mathcal{A}}_{\mathbb{T}}(O)$		$\overline{\mathcal{A}}_{\mathbb{T}}(P)$	
				Time	Ans.	Time	Ans.	Time	Ans.
`half_2.c`	`invgen`	1	25	0.008	×	0.014	≃	0.017	✓
`seq.c`	`invgen`	2	30	0.015	×	0.084	✓	0.045	✓
`sum01*.c`	`loops`	1	15	0.008	×	0.009	✓	0.041	✓
`count_up_d*.c`	`loops`	1	15	0.002	×	0.008	≃	0.011	✓
`hhk2008.c`	`lit`	2	20	0.003	×	0.073	≃	0.032	✓
`gsv2008.c`	`lit`	1	20	0.002	×	0.007	✓	0.015	✓
`Mysore.c`	`crafted`	1	30	0.0008	×	0.002	✓	0.004	✓
`Copenhagen.c`	`crafted`	2	30	0.002	×	0.012	≃	0.021	✓

Although for all benchmarks $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ infers more precise invariants, still $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ also takes more time than $\mathcal{A}(P)$, as expected. On our benchmarks, this translates to slow-downs (i.e., $\mathcal{A}(P)$ vs. $\overline{\mathcal{A}}_{\mathbb{T}}(P)$) of 4.9 times in average when $|\mathbb{F}| = 1$, and of 6.9 times in average when $|\mathbb{F}| = 2$. However, in some cases the more efficient version $\overline{\mathcal{A}}_{\mathbb{T}}(O)$, which uses octagons, can also provide more precise results than $\mathcal{A}(P)$. For example, $\overline{\mathcal{A}}_{\mathbb{T}}(O)$ for `half_2.c` gives the precise "erroneous" answer like $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ but gives "I don't know" in all other cases, whereas $\overline{\mathcal{A}}_{\mathbb{T}}(O)$ for `count_up_down*.c` gives the precise "erroneous" and "unreachable" answers like $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ but it turns the "correct" answer from $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ into an "I don't know". On the other hand, for `gsv2008.c` and `Mysore.c`, $\overline{\mathcal{A}}_{\mathbb{T}}(O)$ gives the same precise answers as $\overline{\mathcal{A}}_{\mathbb{T}}(P)$, but twice faster. Furthermore, for `sum01*.c`, even $\overline{\mathcal{A}}_{\mathbb{T}}(I)$, which uses intervals, gives the same precise answers like $\overline{\mathcal{A}}_{\mathbb{T}}(P)$, but with the similar time performance as $\mathcal{A}(P)$. Table 2 shows the running times of $\mathcal{A}(P)$, $\overline{\mathcal{A}}_{\mathbb{T}}(O)$, and $\overline{\mathcal{A}}_{\mathbb{T}}(P)$, as well as whether the corresponding analysis precisely evaluates the given assertion – denoted by Ans. (we use ✓ for yes, ≃ for partially yes, and × for no).

Discussion

Our experiments demonstrate that the lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$ is able to infer more precise numerical invariants than the single-program analysis $\mathcal{A}(\mathbb{D})$ while maintaining scalability (addresses **RQ1**). As the result of more complex abstract operations and transfer functions of the decision tree domain, we observe slower running times of $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$ as compared to $\mathcal{A}(\mathbb{D})$. However, this is an acceptable precision/cost tradeoff, since the more precise numerical invariants inferred by $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$ enables us to successfully answer many interesting assertions in all considered benchmarks (addresses **RQ2** and **RQ3**). Furthermore, our current tool is only a prototype implementation to experimentally confirm the suitability of our approach. Many abstract operations and transfer functions of the lifted domain can be further optimized, thus making the performances of the tool to improve.

Our current tool supports a non-trivial subset of C, and the missing constructs (e.g. pointers, `struct` and `union` types) are largely orthogonal to the solution (lifted domains). In particular, these features complicate the abstract semantics of single-programs and implementation of the domains for leaf nodes, but have no impact on the semantics of variability-specific constructs and the lifted domains we introduce in this work. Therefore, supporting these constructs would not provide any new insights to our evaluation. If a real-world tool based on abstract interpretation (e.g. ASTREE [14]) becomes freely available, we can easily transfer our implementation to it.

6 Related Work

Decision-tree abstract domains have been a topic of research in the field of abstract interpretation in recent times [25, 15, 9, 46]. Decision trees have been applied for the disjunctive refinement of interval (boxes) domain [25]. That is, each element of the new domain is a propositional formula over interval linear constraints. Decision tree abstract domains has also been used to enable path dependent static analysis [15, 9] by handling disjunctive analysis properties. Binary decision tree domains [9] can express disjunctive properties depending on the boolean values of the branch (`if`) conditions (represented in decision nodes) with sharing of the properties of the other variables (represented in leaf nodes). Segmented decision tree abstract domains [15] are generalizations of binary decision tree domains and array segmentation, where the choices in decision nodes are made on the values of decision variables according to the ranges specified by a symbolic segmentation. A pre-analysis is used to find decision variables and their symbolic segmentation. The choices for a given decision variable are made only once along a given path. The decision tree lifted domain proposed here can be considered as a generalization of the segmented decision tree domain, where the choices for a given feature variable can be made several times along a given path and arbitrary linear constraints over feature variables can be used to represent the choices in decision nodes. Moreover, linear constraints labelling decision nodes here are semantically inferred during the static analysis and do not necessarily syntactically appear in the code. Urban and Mine [46] use decision tree-based abstract domains to prove program termination. Decision nodes are labelled with linear constraints that split the memory space and leaf nodes contain affine ranking functions for proving program termination. The APRON library has been developed by Jeannet and Mine [33] to support the application of numerical abstract domains in static analysis. The ELINA library [44] represents an another efficient implementation of numerical abstract domains.

Several *lifted analyses based on abstract interpretation* have been proposed recently [36, 23, 18, 19, 21] for analyzing traditional program families with `#ifdef`-s. A formal methodology for derivation of tuple-based lifted analyses from existing single-program analyses phrased in the abstract interpretation framework has been proposed by Midtgaard et. al. [36]. They use a lifted domain that is a $|\mathbb{K}|$-fold product of an existing single-program domain. That is, the elements of the lifted domain are tuples that contain one separate component for each configuration of $\mathbb{K}$. A more efficient lifted analysis by abstract interpretation obtained by improving representation via BDD-based lifted domains is proposed by Dimovski [18, 19]. The elements of the lifted domain are BDDs, in which decision nodes are labelled with Boolean features and leaf nodes belong to an existing single-program domain. BDDs offer more possibilities for sharing and interaction between analysis properties corresponding to different configurations. The above lifted analyses are applied to program families with only Boolean features. The work [21] extends prior approaches by using decision tree-based lifted domain for analyzing program families with numerical features. In this case, the elements of the lifted domain are decision trees, in which decision nodes are labelled with linear constraints over numerical features and leaf nodes belong to an existing single-program domain. This domain is also successfully applied to program synthesis for resolving program sketches [22]. Several other efficient implementations of the lifted dataflow analysis from *the monotone framework* (a-la Kildall) [35] have also been proposed in the SPL community. Brabrand et al. [5] have introduced a tuple-based lifted dataflow analysis, whereas an approach based on using variational data structures (e.g., variational CFGs, variational data-flow facts) [47] have been used for achieving efficient dataflow computation of some real-world systems.

Finally, SPL$^{\text{LIFT}}$ [4] is an implementation of the lifted dataflow analysis formulated within the IFDS framework, which is a subset of dataflow analyses with certain properties, such as distributivity of transfer functions.

Dynamic program families (DSPLs) have been introduced by Hallsteinsen et al. [28] in 2008 as a technique that uses the principles of traditional SPLs to build variants adaptable at run-time. Since then, the research on DSPLs has been mainly focussed on developing mechanisms for implementing DSPLs and for defining suitable feature models.

There are many strategies for implementing variability in traditional SPLs, such as: annotative approach via the C-preprocessor's `#ifdef` construct [34], compositional approach via the feature-oriented programming (FOP) [40] and the delta-oriented programming (DOP) [43], etc. The extensions of FOP and DOP to support run-time reconfiguration and software evolution as found in DSPLs has been proposed by Rosenmuller et al. [42] and Damiani et al [17]. In this work, we extend the annotative approach via `#ifdef`-s to implement variability in DSPLs. The set of valid configurations $\mathbb{K}$ of a program family with Boolean and numerical features is typically described by a *numerical feature model*, which represents a tree-like structure that describes which combinations of feature's values and relationships among them are valid. Several works address the need to change the structural variability (feature model) at run-time. One approach [30] relies on the Common Variability Language (CVL) as an attempt for modelling variability transformations by allowing different types of substitutions to re-configure new versions of base models. Cetina et al. [8] also propose several strategies for modelling runtime transformations using CVL. Helleboogh et al. [31] use a meta-variability model to support dynamic feature models, where high-level constructs enable the addition and removal of variants on-the-fly to the base feature model. In this work, we disregard syntactic representations of the set $\mathbb{K}$ as feature model, as we are concerned with behavioural analysis of program families rather than with implementation details of $\mathbb{K}$. Therefore, we use the set-theoretic view of $\mathbb{K}$ that is syntactically fixed a priori. This is convenient for our purpose here. To the best of our knowledge, our work is pioneering in studying specifically designed behavioral analysis of dynamic program families.

7 Conclusion

In this work, we employ decision trees and widely-known numerical abstract domains for the automatic analysis of C program families that contain dynamically bound features. This way, we obtain a decision tree lifted domain for handling dynamic program families with numerical features. Based on a number of experiments on benchmarks from SV-COMP, we have shown that our lifted analysis is effective and performs well on a wide variety of cases by achieving a good precision/cots tradeoff. The lifted domain $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D})$ is very expressive since it can express weak forms of disjunctions arising from feature-based constructs.

In the future, we would like to extend the lifted abstract domain to also support non-linear constraints, such as congruences and non-linear functions (e.g. polynomials, exponentials) [6]. Note that the lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$ reports constraints defined over features for which a given assertion is valid, fails, or unreachable. The found constraints take into account the value of features at the location before the given assertion. By using a backward lifted analysis [24, 38], which propagates backwards the found constraints by $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$, we can infer the necessary preconditions (defined over features) in the initial state that will guarantee the assertion is always valid, fails, or unreachable.

References

1 Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013. doi:10.1007/978-3-642-37521-7.

2 Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 372–375, 2011. doi:10.1109/ASE.2011.6100075.

3 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. doi:10.1109/ICSE.2013.6606594.

4 Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spl$^{\text{lift}}$: statically analyzing software product lines in minutes instead of years. In *ACM SIGPLAN Conference on PLDI '13*, pages 355–364, 2013. doi:10.1145/2491956.2491976.

5 Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. *T. Aspect-Oriented Software Development*, 10:73–108, 2013. doi:10.1007/978-3-642-36964-3_3.

6 Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, volume 3580 of *LNCS*, pages 1349–1361. Springer, 2005. doi:10.1007/11523468_109.

7 Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz Cortés, and Mike Hinchey. An overview of dynamic software product line architectures and techniques: Observations from research and industry. *J. Syst. Softw.*, 91:3–23, 2014. doi:10.1016/j.jss.2013.12.038.

8 Carlos Cetina, Øystein Haugen, Xiaorui Zhang, Franck Fleurey, and Vicente Pelechano. Strategies for variability transformation at run-time. In *Software Product Lines, 13th International Conference, SPLC 2009, Proceedings*, volume 446 of *ACM International Conference Proceeding Series*, pages 61–70. ACM, 2009. URL: https://dl.acm.org/citation.cfm?id=1753245.

9 Junjie Chen and Patrick Cousot. A binary decision tree abstract domain functor. In *Static Analysis - 22nd International Symposium, SAS 2015, Proceedings*, volume 9291 of *LNCS*, pages 36–53. Springer, 2015. doi:10.1007/978-3-662-48288-9_3.

10 Andreas Classen, Arnaud Hubaux, and Patrick Heymans. A formal semantics for multi-level staged configuration. In *Third International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain, January 28-30, 2009. Proceedings*, volume 29 of *ICB Research Report*, pages 51–60. Universität Duisburg-Essen, 2009. URL: http://www.vamos-workshop.net/proceedings/VaMoS_2009_Proceedings.pdf.

11 Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

12 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

13 Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Proceedings*, volume 631 of *LNCS*, pages 269–295. Springer, 1992. doi:10.1007/3-540-55844-6_142.

14 Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Proceedings*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005. doi:10.1007/978-3-540-31987-0_3.

15 Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. A scalable segmented decision tree abstract domain. In *Time for Verification, Essays in Memory of Amir Pnueli*, volume 6200 of *LNCS*, pages 72–95. Springer, 2010. doi:10.1007/978-3-642-13754-9_5.

16 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–96. ACM Press, 1978. doi:10.1145/512760.512770.

17 Ferruccio Damiani, Luca Padovani, and Ina Schaefer. A formal foundation for dynamic delta-oriented software product lines. In *Generative Programming and Component Engineering, GPCE'12*, pages 1–10. ACM, 2012. doi:10.1145/2371401.2371403.

18 Aleksandar S. Dimovski. Lifted static analysis using a binary decision diagram abstract domain. In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019*, pages 102–114. ACM, 2019. doi:10.1145/3357765.3359518.

19 Aleksandar S. Dimovski. A binary decision diagram lifted domain for analyzing program families. *Journal of Computer Languages*, 63:101032, 2021. doi:10.1016/j.cola.2021.101032.

20 Aleksandar S. Dimovski and Sven Apel. Tool artifact for "lifted static analysis of dynamic program families by abstract interpretation". *Zenodo*, 2021. doi:10.5281/zenodo.4718697.

21 Aleksandar S. Dimovski, Sven Apel, and Axel Legay. A decision tree lifted domain for analyzing program families with numerical features. In *Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Proceedings*, volume 12649 of *LNCS*, pages 67–86. Springer, 2021. doi:10.1007/978-3-030-71500-7_4.

22 Aleksandar S. Dimovski, Sven Apel, and Axel Legay. Program sketching using lifted analysis for numerical program families. In *NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings*, volume 12673 of *LNCS*. Springer, 2021.

23 Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Variability abstractions: Trading precision for speed in family-based analyses. In *29th European Conference on Object-Oriented Programming, ECOOP 2015*, volume 37 of *LIPIcs*, pages 247–270. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.ECOOP.2015.247.

24 Aleksandar S. Dimovski and Axel Legay. Computing program reliability using forward-backward precondition analysis and model counting. In *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Proceedings*, volume 12076 of *LNCS*, pages 182–202. Springer, 2020. doi:10.1007/978-3-030-45234-6_9.

25 Arie Gurfinkel and Sagar Chaki. Boxes: A symbolic abstract domain of boxes. In *Static Analysis - 17th International Symposium, SAS 2010. Proceedings*, volume 6337 of *LNCS*, pages 287–303. Springer, 2010. doi:10.1007/978-3-642-15769-1_18.

26 Robert J. Hall. Fundamental nonmodularity in electronic mail. *Automated Software Engineering*, 12(1):41–79, 2005. doi:10.1023/B:AUSE.0000049208.84702.84.

27 Svein O. Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2nd international workshop on dynamic software product lines DSPL 2008. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*, page 381. IEEE Computer Society, 2008. doi:10.1109/SPLC.2008.69.

28 Svein O. Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008. doi:10.1109/MC.2008.123.

29 Svein O. Hallsteinsen, Erlend Stav, Arnor Solberg, and Jacqueline Floch. Using product line techniques to build adaptive systems. In *Software Product Lines, 10th International Conference, SPLC 2006, Baltimore, Maryland, USA, August 21-24, 2006, Proceedings*, pages 141–150. IEEE Computer Society, 2006. doi:10.1109/SPLINE.2006.1691586.

30 Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. Adding standardized variability to domain specific languages. In *Software Product Lines, 12th*

International Conference, SPLC 2008, Proceedings, pages 139–148. IEEE Computer Society, 2008. doi:10.1109/SPLC.2008.25.

31 Alexander Helleboogh, Danny Weyns, Klaus Schmid, Tom Holvoet, Kurt Schelfthout, and Wim Van Betsbrugge. Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines. In *Synamic Software Product Lines, 3rd International Workshop, SSPL 2009, Proceedings*, 2009.

32 Alexandru F. Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Effective analysis of c programs by rewriting variability. *Programming Journal*, 1(1):1, 2017. doi:10.22152/programming-journal.org/2017/1/1.

33 Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification, 21st International Conference, CAV 2009. Proceedings*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009. doi:10.1007/978-3-642-02658-4_52.

34 Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, Germany, May 2010.

35 Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages, (POPL'73)*, pages 194–206, 1973. doi:10.1145/512927.512945.

36 Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Systematic derivation of correct variability-aware program analyses. *Sci. Comput. Program.*, 105:145–170, 2015. doi:10.1016/j.scico.2015.04.005.

37 Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006. doi:10.1007/s10990-006-8609-1.

38 Antoine Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages*, 4(3-4):120–372, 2017. doi:10.1561/2500000034.

39 David Lorge Parnas. On the design and development of program families. *IEEE Trans. Software Eng.*, 2(1):1–9, 1976. doi:10.1109/TSE.1976.233797.

40 Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP'97 - Object-Oriented Programming, 11th European Conference, 1997, Proceedings*, volume 1241 of *LNCS*, pages 419–443. Springer, 1997. doi:10.1007/BFb0053389.

41 Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. Flexible feature binding in software product lines. *Autom. Softw. Eng.*, 18(2):163–197, 2011. doi:10.1007/s10515-011-0080-5.

42 Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. Tailoring dynamic software product lines. In *Generative Programming And Component Engineering, Proceedings of the 10th International Conference on Generative Programming and Component Engineering, GPCE 2011*, pages 3–12. ACM, 2011. doi:10.1145/2047862.2047866.

43 Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010. Proceedings*, volume 6287 of *LNCS*, pages 77–91. Springer, 2010. doi:10.1007/978-3-642-15579-6_6.

44 Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Making numerical program analysis fast. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015*, pages 303–313. ACM, 2015. doi:10.1145/2737924.2738000.

45 Caterina Urban. *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs*. PhD thesis, École Normale Supérieure, Paris, France, 2015. URL: https://tel.archives-ouvertes.fr/tel-01176641.

46 Caterina Urban and Antoine Miné. A decision tree abstract domain for proving conditional termination. In *Static Analysis - 21st International Symposium, SAS 2014. Proceedings*, volume 8723 of *LNCS*, pages 302–318. Springer, 2014. doi:10.1007/978-3-319-10936-7_19.

47 Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. Variability-aware static analysis at scale: An empirical study. *ACM Trans. Softw. Eng. Methodol.*, 27(4):18:1–18:33, 2018. doi:10.1145/3280986.

48 Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. Variability encoding: From compile-time to load-time variability. *J. Log. Algebraic Methods Program.*, 85(1):125–145, 2016. doi:10.1016/j.jlamp.2015.06.007.

Best-Effort Lazy Evaluation for Python Software Built on APIs

Guoqiang Zhang
Department of Computer Science, North Carolina State University, Raleigh, NC, USA

Xipeng Shen
Department of Computer Science, North Carolina State University, Raleigh, NC, USA

Abstract

This paper focuses on an important optimization opportunity in Python-hosted domain-specific languages (DSLs): the use of laziness for optimization, whereby multiple API calls are deferred and then optimized prior to execution (rather than executing eagerly, which would require executing each call in isolation). In existing supports of lazy evaluation, laziness is "terminated" as soon as control passes back to the host language in any way, limiting opportunities for optimization. This paper presents Cunctator, a framework that extends this laziness to more of the Python language, allowing intermediate values from DSLs like NumPy or Pandas to flow back to the host Python code without triggering evaluation. This exposes more opportunities for optimization and, more generally, allows for larger computation graphs to be built, producing 1.03-14.2X speedups on a set of programs in common libraries and frameworks.

2012 ACM Subject Classification Software and its engineering → Dynamic compilers; Software and its engineering → Runtime environments

Keywords and phrases Lazy Evaluation, Python, API Optimization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.15

Funding This material is based upon work supported by the National Science Foundation (NSF) under Grants CCF-1703487. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

1 Introduction

Modern software is built upon APIs. Although APIs typically encapsulate highly optimized code, suboptimal usage of APIs can cause large performance degradation. Such a problem is especially common in Python programs, as Python has become the host language of many popular libraries or domain-specific languages (DSL) targeting performance-demanding tasks, such as NumPy [28], Pandas [17], PySpark [31], TensorFlow [1], and PyTorch [25].

```
S1: x = numpy.add(a,b)
S2: x = numpy.add(x,c)
```

(a)

```
S1: wa= weldarray(a)
S2: x = weldnumpy.add(wa,b)
S3: x = weldnumpy.add(x,c)
S4: x.evaluate()
```

(b)

```
S1: wa = weldarray(a)
S2: x = weldnumpy.add(wa,b)
S3: x = weldnumpy.add(x,c)
S4: a[0] = 0
S5: x.evaluate()
```

(c)

Figure 1 NumPy example and WeldNumpy variants.

Suboptimal usage of APIs typically involves a sequence of API calls. For illustration purpose, we show a simple example in Figure 1(a). While the code is simple, it suffers the performance flaw of a redundant temporary value: S1 creates an object and assigns it to `x`, but after `x` points to another object in S2, Python garbage collector (GC) releases the former object as it now has zero reference count. The program can be optimized by replacing the second statement with an in-place operation: `numpy.add(x, c, out=x)`. The argument `out=x` instructs `numpy.add` to reuse `x` to store the result. The optimization not only improves data locality, but also reduces memory usage.

Existing work to tackle the problem of suboptimal API sequence relies on lazy evaluation. Several API sets, such as Spark [31], TensorFlow [1], and WeldNumpy [22], have been designed and implemented in that way. They designate some APIs as eager APIs and the rest as lazy APIs. Invocations of lazy APIs only log the APIs in a certain form rather than execute them. Once an eager API is encountered, the logged sequence of APIs will be optimized together and then executed. For instance, Figure 1b shows the WeldNumpy version of the code in Figure 1a; the two `add` operations are not evaluated until S4; before the evaluation happens, the WeldNumpy runtime optimizes the two add operations and avoids the unnecessary object creation for `x` in the second add operation.

A fundamental problem underlying the API-based lazy evaluation is the data dependence that arises between the invocations of the APIs and the host Python code. Figure 1c gives an illustration. Compared to Figure 1b, the difference is that a Python statement S4 updates the input of S1 before `evaluate()`. Python statements, by default, are eagerly evaluated. But as the `weldnumpy.add` API is lazily evaluated, S2 would end up using the wrong values of `a`.

Existing frameworks either leave the issue to the programmers (e.g., in WeldNumpy [22]), relying on them to put in eager APIs at the right places, or design the library such that any API that might incur dependencies with the host code is designated as an eager API, regardless of the context (e.g., in Spark [31] or TensorFlow [1]). The former increases programmers' burdens, while the latter often misses optimization opportunities due to its conservative design.

Listing 1 shows an example in *Spark*. It loads a text file (Line 1), splits the lines into words (Line 2), filters out illegal words (Line 3), counts the number of words (Line 4), sums the lengths of all words (Line 5), and finally outputs the average word length (Line 5). In *Spark*, the APIs `textFile`, `flatMap`, `filter`, and `map` are always lazily evaluated; both `count` and `sum` are always eagerly evaluated APIs because they return values to the host code and hence the value, in general, could potentially be operated on by the host code. When an eager API is invoked, Spark fuses relevant lazy APIs together into a pipeline; intermediate results are not cached. As there are two eager API calls, the lazy operations `textFile`, `filter`, and `flatMap` are evaluated twice at lines 4 and 5. The solution from *Spark* is to introduce extra APIs such that programmers can use them for caching. This "band-aid" solution further increases the burdens of programmers, who now need to be concerned of not only the usage of the many existing APIs but also the best places to use the caching APIs.

Our study (§9) shows that these limitations prevent existing frameworks from tapping into the full potential of lazy evaluations for Python+API programs, leaving up to 14X performance improvement yet to harvest.

The primary goal of this work is to create a solution that overcomes the limitations of the existing methods for enabling lazy evaluation for Python+API programming. The principles for developing our solution are two fold: (1) It should be automatic such that programmers do not need to worry about manually finding the best places in their code to insert APIs to trigger evaluations; (2) it should be effective in postponing API evaluations to places as late as possible to maximize API optimization opportunities.

Listing 1 A Spark program with performance issues that are hard to automatically optimize away

```
1 lines = sc.textFile("foo")
2 ws = lines.flatMap(lambda l: l.split())
3 ws = ws.filter(lambda x: re.match("^[\w]+$", x))
4 word_count = ws.count()
5 total_len = ws.map(lambda w: len(w)).sum()
6 avg_len = total_len / word_count
```

The key to both principles is to effectively analyze data dependencies between the host code and the APIs in a Python program. The problem is challenging. Many features of Python, such as dynamic typing and reflection, make analysis of the host code difficult. The difficulty is exacerbated by the extra need to analyze library APIs and their interactions with the host code. The lack of such automatic data dependence analysis is plausibly one of the main reasons for the unsatisfying solutions being used today.

In this work, we address the challenge by developing a *minimum interference runtime watching scheme* (MIN-watch for short). The basic idea underlying MIN-watch is simple, tracking data accesses at runtime to detect data dependencies. The novelty is in how MIN-watch makes the tracking efficient and effective for sound dependence detection in the context of Python+API programs. MIN-watch does it by taking advantage of the characteristics of Python and the special needs in lazy evaluation for Python+API. It is resilient to Python language complexities. It minimizes runtime overhead through a focused tracking scope in data and an efficient runtime checking mechanism (bit-level flagging and deferred flag resetting). It meanwhile imposes near-zero burdens on programmers. MIN-watch is based on a dependence theorem we introduce to formulate the correctness of lazy evaluation in this host+API context (§3).

Based on MIN-watch, we further develop Cunctator, a software framework for materializing the extended lazy evaluation. Cunctator consists of an intermediate representation (lazy IR) for the deferred operations, a lazy IR evaluator, a class that delegates the results of deferred operations and postpones operations applied to itself, and a set of interfaces for redirecting API calls and registering optimizers. With these components together, Cunctator provides programmers the conveniences of enabling the automatic *Best-Effort Lazy Evaluation (BELE)* for a Python library and harvesting the optimization benefits.

To demonstrate the usefulness of Cunctator, we implement four optimizations enabled by BELE for three API packages (`numpy, Spark, Pandas`). Experiments on 15 programs show that the optimizations generate 1.03-14.2X speedups. Stress testing shows that the overhead of Cunctator is no greater than 2.25% (in its default setting).

In summary, this work makes the following major contributions:

- It introduces the concept of Best-Effort Lazy Evaluation, and shows that MIN-watch is effective in enabling data dependence analysis for Python+API programs to support Best-Effort Lazy Evaluation.
- It develops the first software framework to support Best-Effort Lazy Evaluation for Python+API programs.
- It demonstrates the effectiveness of the techniques in enabling optimizations of Python+API programs.

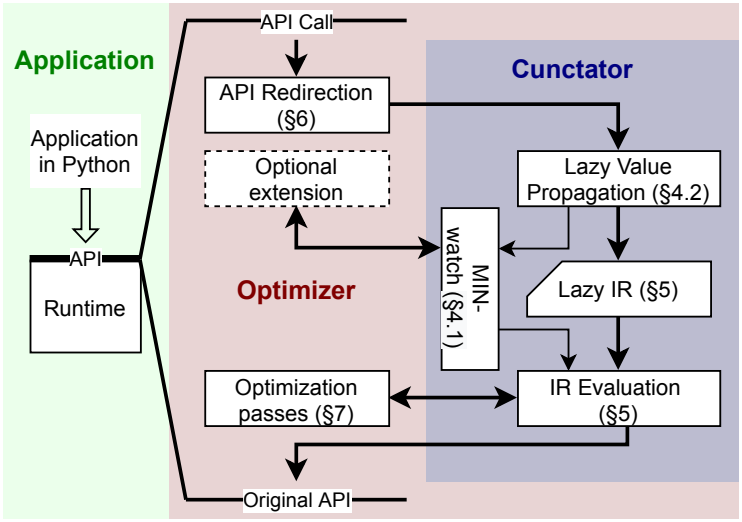


Figure 2 Architecture of Cunctator.

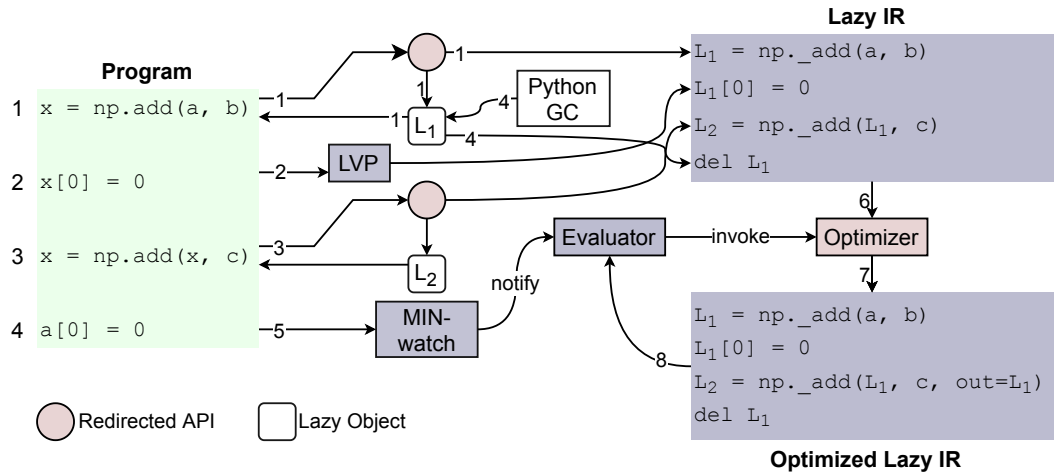


Figure 3 Running example. Numbers on directed edges indicate the order of actions.

2 Overview

Figure 2 illustrates Cunctator's architecture. When an application invokes a DSL API, the API call is redirected to a Cunctator optimizer. Instead of evaluating the API, the optimizer records the API in the form of Lazy IR (§5), and returns a *lazy object*. The lazy object supports *Lazy Value Propagation* (LVP, see §4.2), which tries to propagate a new lazy object when an operation is applied to the lazy object. Cunctator employs MIN-watch (§4.1) to monitor accesses to objects related with the deferred operations. When MIN-watch encounters host statements or APIs that prevent further delays (based on dependence theorems in §3), it triggers the evaluation of the deferred operations. During the evaluation, the Cunctator optimizer applies optimization passes (§6) onto the IR, and then invokes the original DSL APIs for evaluation. To apply Cunctator to a domain, the developers of the DSL optimizer only needs to use Cunctator interfaces to specify redirections of the domain APIs, to support MIN-watch for some common types, and to write domain-specific optimizations. The extra work an application developer needs to do is just to import one or several modules.

Figure 3 shows the execution flow of a NumPy program with Cunctator. First, the `np.add` in line 1 is redirected to Cunctator optimizer, which records the API call as a lazy IR instruction and returns a lazy object L_1. Note, the assignment to x is not deferred but executed, and x now points to the lazy object L_1. The optimizer also sets up the two arguments, `a` and `b`, for watching. At line 2, because `x` is lazy, the Lazy Object class automatically captures and logs this operation and defers its execution. Line 3 is similar to line 1, and Cunctator defers and logs the operation. That assignment makes `x` point to L_2; L_1's reference count reduces to zero, which triggers Python's garbage collection on L_1. L_1's deconstructor, however, rather than deconstructs L_1, defers the deconstruction and inserts a `del` instruction into the IR. Line 4 tries to update `a`, which is captured by MIN-watch, which triggers the evaluation of all the deferred operations. The evaluator first invokes the optimizer, which reduces redundant temporary variables, and then evaluates the operations.

Before presenting the details of Cunctator, in the next section, we first define some terms and prove a dependence theorem that formulate Cunctator's correctness.

3 Dependencies between Operations

To ensure the correctness of Cunctator, one key aspect is to properly manage the dependencies between postponed API calls and eagerly executed statements. We first introduce a set of terms that are used in the following discussions.

Terminology. Unless otherwise stated, an *object* denotes a Python object. An *operator* denotes a Python built-in operator. An *operation* denotes the process of applying an operator to its operands. For example, `foo.bar()` consists of two *operations*: The '.' *operator* is applied to `foo` and `"bar"` to return a function object, which becomes the operand of the '()' *operator*. We in addition introduce the following terms.

- *Contents of an object*: All in-memory states that could be potentially accessed or updated *directly* through the object's fields and methods. Take the `list` object `["foo", "bar"]` as an example – its contents are the references to its two elements, the string objects, rather than the characters of the strings. By this definition, two objects could share contents, namely, their methods or attributes could access or update the same in-memory state.
- *Sealed object*: An object that shares no content with other objects. This means the contents of a sealed object can be accessed or updated only by its own attributes or methods.
- *Domestic object*: An object whose attributes and methods access no external resources (e.g., files and network) but only the memory space of the current process. We are interested in sealed and domestic objects (e.g., `list` objects).
- *Dependents of an object*: The object itself and the objects referred to in the contents of the object. For example, the dependents of a `list` are itself and its elements.
- *Relatives of an object*: Object R is a relative of object O if and only if there is a dependent chain $O \leftarrow ... \leftarrow R$, in which $x \leftarrow y$ denotes that object y is a dependent of object x.
- *Regular operation*: An operation is regular if the relatives of its operands and return value are all sealed and domestic, and it only accesses or updates the contents of its operands' relatives or newly created objects during the operation. In most cases, a DSL API call is a regular operation. One example of irregular operation is an API manipulating global variables.

Without noting otherwise, the following discussions assume regular operations and sealed and domestic objects and there is no exceptions. Section 4.4 discusses exceptions and other complexities.

Dependency types. Based on the above definitions, we classify potential dependencies between an API call O_A and a statement O_B into three types:

- *Return-access*: The return value of O_A is accessed (read or written) by O_B, as illustrated by the top left example in Figure 4.
- *Input-update*: A relative of O_A's operands is updated in O_B, illustrated by the bottom left example in Figure 4.
- *Update-access*: O_A updates a relative of its operand I and O_B accesses that relative, illustrated on the right side in Figure 4. Cunctator uses a conservative version of this definition, which forgoes the requirement of the two relatives being the same. It simplifies runtime checking as shown later.

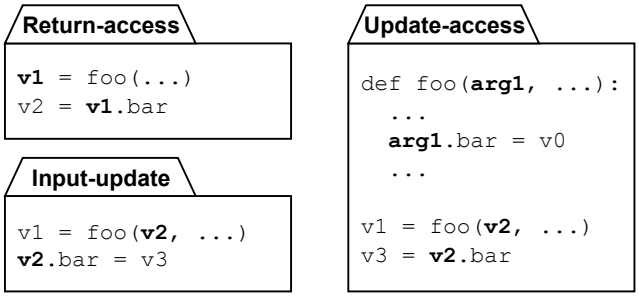


Figure 4 Three types of dependencies.

Dependency Theorem. This part presents the dependence theorem governing the validity of lazy evaluation for APIs, which underpins the design of Best-Effort Lazy Evaluation.

▶ **Lemma 1.** *For an API call A followed by a statement B, deferring the execution of A to a point after B does not change the data dependencies between them if there are no* return-access*,* input-update*, or* update-access *dependencies between them.*

The lemma comes from the observation that for the properties of sealed and domestic objects and regular operations, the three types of dependencies cover all possible data dependencies (true dependencies, anti-dependencies, output dependencies) [2] between two statements.

▶ **Theorem 2.** *For an API call A followed by a sequence of statements S, deferring the execution of A to a point after S is valid if there are no* return-access*,* input-update*, or* update-access *dependencies between A and any of the statements in S.*

This theorem is derived from the classic *fundamental theorem of dependence* [2], which states the following: Any reordering transformation that preserves every dependence in a program preserves the meaning of that program. Deferring executions is clearly a kind of reordering transformation. The deferring does not cause any dependence changes according to Lemma 1 for none of the three types of dependencies exist between A and S. The theorem hence holds.

Theorem 2 is essentially a variant of the *fundamental theorem of dependence* in the context of API lazy evaluation; the benefits of having it are however significant. It entails what types of dependencies are needed to consider during lazy evaluation, and what set of data objects are needed to watch, which lay the foundation for the design of MIN-watch and BELE in the next section.

4 Best-Effort Lazy Evaluation (BELE)

The purpose of BELE is to defer DSL API calls until the necessary moment. The central challenge that BELE confronts is to satisfy three mutually constrained requirements: First, BELE has to ensure correctness of the program. Second, the deferring period, or *laziness*, should be as long as possible to harvest optimization opportunities. Finally, the overhead should be low.

To address these challenges, we introduce minimum interference runtime watching (MIN-watch) in §4.1 to detect, with low overhead, *input-update* and *update-access* dependencies between deferred API calls and host code. In addition, Cunctator §4.2 employs *lazy value propagation* (LVP) to manage *return-access* dependencies while ensuring enough laziness. The overheads of Cunctator and strategies to control them are discussed in §4.3. Finally, §4.4 describes how to handle special scenarios.

4.1 Minimum interference runtime watching (MIN-Watch)

Based on Theorem 2, the key for BELE is in detecting data dependencies. MIN-watch takes the way of runtime object watching, which makes it not subject to the language complexities Python imposes on compilers or other static methods.

4.1.1 Overview of MIN-Watch

What makes MIN-watch distinctive over common runtime access tracking is the strategy it employs, which takes advantage of the characteristics of this problem setting and Python language properties, and uses a lightweight type-based scheme for non-intrusive implementation. Specifically, the design of MIN-watch draws on three observations: (1) In Python, most memory accesses go through object interface with multiple layers of redirection and procedure abstractions, hence a much reduced sensitivity to runtime memory access tracking overhead compared to many other languages and settings. (2) The key to BELE is the dependence between API and host. So many data accesses that are irrelevant to such dependencies can be exempted from tracking. (3) Python object assignments and parameter passing are both through references; so to check dependencies related to an actual object, it is not necessary to track references to it, if the watching scheme is put onto that object.

Built on the observations, MIN-watch has the following features: (1) By focusing on API to host dependencies and Theorem 2, MIN-watch concentrates runtime watching on only relevant data objects. (2) It employs an efficient runtime checking mechanism (bit-level flagging and deferred flag resetting) via the Python interpreters to minimize interference to program executions. (3) It employs a type-based approach to enabling runtime object watching, but does it in a non-intrusive way such that application developers need to make no changes to the implementation of a data type for the approach to take effect. Moreover,

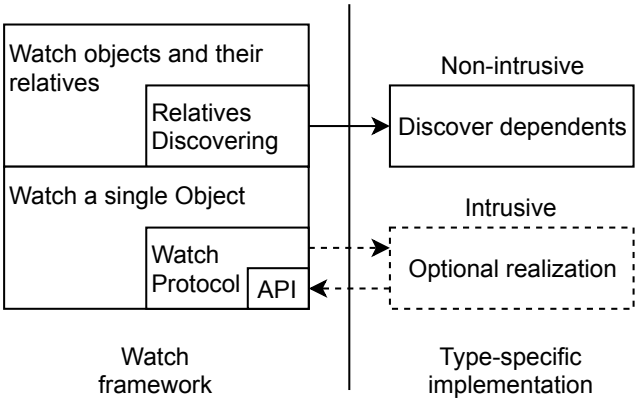


Figure 5 The architecture of MIN-watch. Arrows indicate invocation; dotted lines indicate optional implementation.

```
# in module numpy
# gIR: the global IR scratchpad
def add(a, b):
   setupWatch(a, True)
   setupWatch(b, True)
   id = gIR.add_call(...)
   return Lazy(gIR, id)

def setupWatch(obj, watchUpdateOnly):
   for r in findRelatives(obj):
      r.__set_watch__(watchUpdateOnly)
```

Figure 6 A high-level illustration of how MIN-watch works for API *numpy.add(a,b)*.

the utilities in Cunctator simplify the work an optimizer developer[1] needs to do to enable MIN-watch (and BELE) for a domain DSL. The first two features make MIN-watch efficient, and the other features make it easy to use.

Figure 5 shows the architecture of MIN-watch, and Figure 6 uses `numpy.add(a, b)` as an example to illustrate at a high level how MIN-watch works. The API was overloaded such that when the API is called in a program, instead of doing the computation of arrays addition, it sets up objects a and b and their relatives for runtime watching via function `setupWatch`. Function `setupWatch` calls a function `findRelatives()` to go through each relative of an object, and calls `__set_watch__` of that object to set it up for runtime watching. The setup process flags some special bits in the object header such that the extended Python interpreter, when executing a statement, can recognize such objects and invoke Cunctator lazy evaluation listener to evaluate deferred operations.

We next explain MIN-watch in detail, starting with the basic watch protocol on a single object (§4.1.2), and moving on to describe the procedure in finding and watching all relatives (§4.1.3).

[1] Please note the differences between an application developer and an optimizer developer.

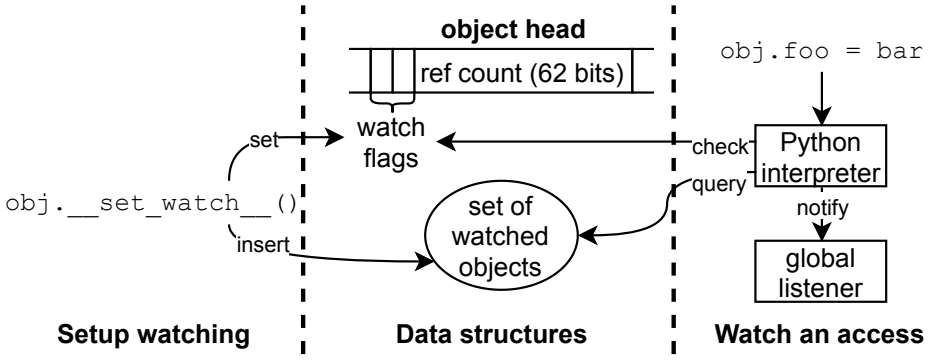


Figure 7 Default implementation of watch protocol.

4.1.2 Watch protocol

Cunctator adds the following method into the root class in Python:

```
def __set_watch__(self, watchUpdateOnly):
   # pseudo-code
   self.watch_flag = WATCH_UPDATE_ONLY
   g_watch_set.insert(self)
```

The parameter *watchUpdateOnly* determines whether the object should be watched for read and write accesses (`false`) or just writes (`true`).

Figure 7 depicts the actions when the protocol takes place. In the setup time (e.g., when `numpy.add(a,b)` is called in the example in Figure 6), `__set_watch__` sets two bits in the object head to indicate whether the object is to be watched for update only (01), read/update (10), or nothing (00). These bits help the interpreter determine the corresponding action quickly. Our implementation borrows the first two bits of the reference count field of an object. That saves extra memory, and also helps ensure that most third-party binary libraries are still compatible by keeping the length of the head unchanged. The method `__set_watch__` in addition adds the object into a global set `watchSet`. It is for fast resetting at the time when the deferred operations are evaluated, which will be elaborated in §4.1.4.

We extend the Python interpreter such that it notifies Cunctator lazy evaluation listener when the content of an object that is being watched is accessed by a bytecode (e.g., `LOAD_ATTR` and `STORE_SUBSCR`).

The default implementation of `__set_watch__` ignores the parameter `watchUpdateOnly` (i.e., assuming it is false). It is because when just encountering the statement, for some data types, the interpreter sometimes cannot tell whether the access would update the object. (Note that it is legitimate in Python for a seemingly read-only (e.g., `foo.bar`) operation to update the object.) This conservative implementation may reduce the laziness but won't cause correctness issues. For a given data type, the optimizer developer can choose to customize its `__set_watch__()` method and other methods to enable a more precise treatment.

It is worth noting that if an object is not sealed, accesses or updates to the object's contents through other objects are not watched with the default implementation. This is fixed in the upper relatives discovering component by not supporting the specific type, which causes the watch process to fail and thus triggers eager evaluations of involved operations.

4.1.3 Watching relatives

With the watch protocol, we can watch a single object. MIN-watch requires watching all the relatives of an object of interest as Figure 6 has shown. The watch framework holds a registry that can register user-defined procedures to discover dependents of specific types. Through recursively calling registered procedures, all relatives of an object can be found. For example, a list A contains objects B, C, and D, and D is another list that contains E and F. Then, the dependent-discovering procedure registered for type `list` returns B, C, and D for object A, after which a recursive call of the procedure for D returns E and F. Typically, a type-specific dependent-discovering procedure is easy to implement. For example, the procedure for `list` is as simple as[2]:

```
def list_deps(l):
    for e in l:
        yield e
```

Algorithm 1 shows the process of setting up to watch an object's relatives. The SetWatch procedure first checks existing watch flags and returns in two cases: The first case is that the object is watched for access, when the procedure returns disregarding the parameter `watchUpdateOnly`. The second case is that the object is watched for updates only and the parameter `watchUpdateOnly` is `True`. In other cases, the procedure sets up to watch the object through the watch protocol and then recursively calls SetWatch for each of its dependents (except for the object itself). If the type of the object is not registered in the registry, the procedure raises an exception, which will be caught by Cunctator to trigger an eager evaluation of the involved operation.

Algorithm 1 Setting up to watch an object's relatives.

```
1:  DepReg ← the registry for discovering dependents
2:  procedure SetWatch(obj, watchUpdateOnly)
3:      if obj is watched for access then return
4:      if watchUpdateOnly ∧ obj is watched then return
5:      obj.___set_watch___(watchUpdateOnly)
6:      for all d ∈ Deps(obj) do
7:          SetWatch(d, watchUpdateOnly)
8:  procedure Deps(obj)
9:      if type(obj) is registered in DepReg then
10:         return DepReg.getHandler(type(obj))(obj)
11:     else
12:         raise an exception
```

When Cunctator defers an operation, it invokes the SetWatch procedure for all the operands except for lazy values, whose potential dependency is handled by lazy value propagation.

[2] `yield` is a Python construct that returns the next value in the next call of its enclosing function.

4.1.4 Unwatching objects via deferred flag resetting

After the deferred operations are triggered to get evaluated (or when a watch procedure is aborted because of an unsupported type, see §4.1.3), Cunctator would need to clear the watch flags of all watched objects. Otherwise, later accesses to them would trigger unnecessary evaluations. Going through all the objects could incur substantial overhead. Cunctator circumvents it by introducing a global *watchSet*. Recall in Figure 7, `__set_watch__()` puts an object to be watched into *watchSet* at setup time. That set is emptied once the evaluation of deferred operations is triggered. Python interpreter, when it encounters an object with watching bits set, would check whether that object is within *watchSet*. If not, it cleans the watch bits; otherwise, it invokes Cunctator lazy evaluation listener.

4.2 Lazy value propagation

Return-access dependency is easy to detect for lazily evaluated operations, since all subsequent visits to the return value fall to the actually returned lazy object, which can trigger the evaluation whenever it is used, similar to how the modifier `lazy` works in some other popular languages (e.g., Scala and Swift). However, too often, a lazy object is used shortly after it is returned. For example, in the statement `(a, b) = lazy_func()`, the lazy object is used to unpack its elements right after it is returned from `lazy_func()`. In such cases, an evaluate-when-used semantics of lazy objects results in short-lived laziness, and leaves no optimization opportunities. As a solution to ensure sufficient laziness, we enhance Python with lazy value propagation (LVP), which propagates new lazy values for most operations applied to existing lazy values. In this way, the return-access dependency is not violated, since the operation that uses the return value is deferred as well.

When a lazy value is being operated, LVP records the operation into the lazy IR and then returns a newly created lazy object. One problem that LVP has to solve is when the propagation should stop – in other words, when the true evaluation should be triggered. An evident scenario is when a lazy value is used to determine the execution branch (e.g., the `if` condition). Theoretically, we could explore all possible paths and collect the lazy IR in the form of computational tree logic (CTL) [4]. Such exploration, however, would introduce large overhead, while its benefit is unclear. Another situation of stopping LVP involves overhead control (see §4.3).

Cunctator implements LVP within the class of lazy objects through operator overwriting, as shown in Listing 2. A lazy object is bound with one lazy IR variable. The `__add__` function, for instance, overwrites the operation `lazy + r`. Their operands are set up for watching before the operation is recorded in the lazy IR. Other operations are overwritten in a similar way, except for the `bool()` operation, which triggers the evaluation, as the operation is invoked when a value is used for branch selection. Although the `bool()` operation does not necessarily imply branching, it is a good heuristic.

A special operation in Python is accessing an object's attribute. Commonly, the attribute is accessed by the '.' notation (e.g. `o.a`), which can be overwritten by `__setattr__()` and `__getattr__()`. But the special `__dict__` attribute can be used to access other attributes. For example, `o.__dict__["a"]` is equivalent to `o.a`. Cunctator extends the Python interpreter to invoke the lazy evaluation listener when the `__dict__` attribute is accessed.

It is worth noting that Cunctator chooses to implement LVP in pure Python for fast prototyping. We plan to re-implement it as a part of Python interpreter in the future. This built-in LVP will have lower overhead and know precisely when a value is used for branch selection.

Listing 2 Lazy value propagation

```
class Lazy:
    def __init__(self, ir, v):
        self.__ir, self.__v = ir, v
    def __add__(self, r):
        if self.__ir.evaluated(self.__v):
            return self.__ir.value(self.__v) + r
        watch(r)
        newv = self.__ir.add_op2('+', self, r)
        return Lazy(self.__ir, newv)
    def __bool__(self):
        return bool(self.__ir.evaluate(self.__v))
    # More overwritten operations
    ...
```

4.3 Overhead control

If there are too many inexpensive DSL API calls or too many propagated lazy values, generating and evaluating the lazy IR could introduce too much overhead. Although such cases never appear in our experiments, we still introduce a dynamic scheme to prevent it from happening in the extreme cases. Cuncator employs a parameter $\mathcal{N}_{IRPS}$ to control how many lazy IR instructions can be generated per second. Initially, Cunctator sets a variable $\mathcal{M}$ to $\mathcal{N}_{IRPS}$. When the total number of generated instructions is equal to $\mathcal{M}$, Cunctator evaluates recorded IR, then it sets $\mathcal{M}$ to $\mathcal{N}_{IRPS} * T$, in which T is the total elapsed time since the first API is deferred. If $\mathcal{M}$'s new value is smaller than its old value, it indicates that the program is an extreme case; Cunctator disables itself by avoding API redirection and LVP. In our experiments, we set $\mathcal{N}_{IRPS}$ to 1000.

4.4 Additional complexities

Exception. Theorem 2 assumes that neither O_A nor O_B raises exceptions. Exceptions could direct the execution to their handlers. If there is no exception handler set up, which is the case for most of the DSL programs we encountered, any raised exception would cause the program to crash. Thus, Cunctator disregards potential exceptions of an operation when there is no installed exception handler. When the current context has exception handlers, Cunctator disables BELE, and thence, all operations are eagerly evaluated. Cunctator checks the currently installed exception handlers through an interface added to the Python interpreter.

External dependency. Theorem 2 assumes that O_A and O_B are not dependent on each other through external resources (e.g., one writes to a file, and the other reads the file). Cunctator considers that the information of whether lazily evaluated APIs access external resources as domain knowledge and relies on the optimizer's developer to provide the knowledge. If none of the lazily evaluated operations access external resources, there is no external dependency. Otherwise, a monitor that watches the program's system calls could notify Cunctator when the program tries to access external resources; Cunctator can then avoid deferring the operations.

Unwatchable objects. Although the watch framework works in most cases, there are objects that cannot be watched because they are not sealed or domestic. For example, if an object holds a segment of shared memory, an update to the shared memory in another process will not notify the listener. In addition, it is impractical to implement MIN-watch for all potential types; thus, some uncommon types may not support MIN-watch. Any kind of unwatchable object causes an involved operation to be eagerly evaluated.

Loss of seal. A sealed object may become not sealed at runtime. For example, `numpy.ones()` creates a sealed object O; however, `O.reshape()` may create a new object P that shares O' data buffer (not a Python object) through pointer alias, rendering that O is not sealed any more, and updates to the data buffer by operating P cannot be monitored by watching O. Therefore, if a type supports MIN-Watch, and there is a method of the type leaks the content, the method needs to mark the involved object as unwatchable by setting watch flags' value to 11 (see §4.1.2). Subsequent attempts to set watch on O will enforce eager evaluation.

5 Intermediate Representation

This section gives details on the design of the lazy IR in Cunctator. The lazy IR has a static single assignment (SSA) form. Each instruction is a 4-tuple:

$$< ID, OP, Operands, Annotation >$$

ID is a globally unique name, which represents the result of current instruction. OP is the operator, such as '+', '.' (attribute access), '[]' (array alike access), '()' (function calls). *Operands* are stored as a list. *Annotation* can be used to store any extra info that the optimizer may use. For an API call, for instance, it is logged as a `call` instruction (OP is '()'), the function pointer is stored in the *Operands* field along with the function's arguments, and the API name is put into the *Annotation* field.

An operand of an IR instruction could be either a lazy value or a non-lazy value. When an operand is a lazy value, the instruction stores its ID. For a non-lazy value, the instruction stores a reference to it. (In our discussion, L_x denotes a lazy value, N_x a non-lazy value, and V_x can be any value.)

Cunctator provides a simple interface for optimizer developers of a DSL to register optimization passes. Each optimization pass accepts a sequence of IR instructions as input, and outputs an optimized sequence. Registered optimization passes are chained in order. During an evaluation, the sequence of all recorded IR instructions since the last evaluation is passed down through all optimization passes.

6 Optimizers

Cunctator is an enabler. By enabling BELE, it paves the way for many optimizations that are not supported by existing DSL frameworks. We have implemented proof-of-concept DSL optimizers for NumPy, Pandas, and Spark. These optimizations fall into two categories: *in-language* optimization and *cross-language* optimization. The in-language optimization tries to identify inefficient API uses and replace them with some other APIs of the DSL. The cross-language optimization tries to replace APIs of one DSL with APIs of another DSL. Two techniques for each category are illustrated in the following sections.

6.1 Reducing temporary variables in NumPy

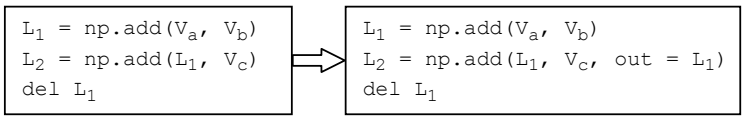


Figure 8 Reducing redundant temporary variables in NumPy.

Redundant temporary variables are a performance issue in many NumPy programs. They impair performance in two ways. First, value assignment to a new variable has worse data locality than an in-place value update. Second, depending on the array size, a temporary variable can consume a lot of memory and thus increase peak memory usage.

When the API call trace is collected as lazy IR in Cunctator, an optimizer can easily optimize away a redundant temporary variable through pattern matching and IR rewriting. At the pattern matching stage, the optimizer locates a redundant temporary variable L_a if the following conditions are all satisfied:

- L_a's value is initialized from the result of an operation that generates a new value rather than performing in-place update.
- L_a participates in no in-place updating operations.
- L_a is passed to an operation O that generates a new value L_b, and O has a counterpart O' that performs an in-place update.
- After being used in operation O, L_a is deleted and participates in no other operations.

At the IR rewriting stage, the optimizer replaces the operation O with O', which saves the result to L_a. Figure 8 shows an example of this optimization technique.

6.2 Adaptive caching for PySpark

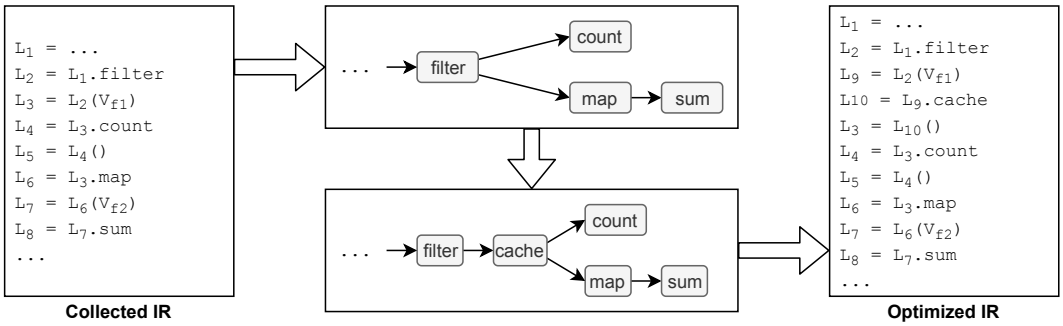


Figure 9 Adding cache operation in Spark.

PySpark is Spark's Python programming interface. Although Spark's runtime employs lazy evaluation to optimize its API call sequences, it fails to handle performance flaws similar to that in Listing 1, because an eager API does not know whether the intermediate result of a lazy API will be used by a subsequent eager API.

With Cunctator, the performance problem in Listing 1 can be optimized away by adding `cache` operations for intermediate results used by more than one eager operation, as shown in Figure 9. The IR shown on the left side of the figure is collected by Cunctator. Note that `del` instructions are omitted for concision. Based on the collected IR, the optimizer constructs a data flow graph for all Spark operations. If two or more eager operations share a common ancestor, the optimizer inserts a `cache` operation at the fork.

Another similar performance problem involves unnecessary `cache` operations, namely, `cache` operations for intermediate results used by only one eager API. Such operations introduce unnecessary memory writing and consume a lot of memory. Based on the same graph analysis as was used for inserting `cache` operations, the optimizer can identify and remove unnecessary `cache` operations.

6.3 From NumPy to WeldNumpy

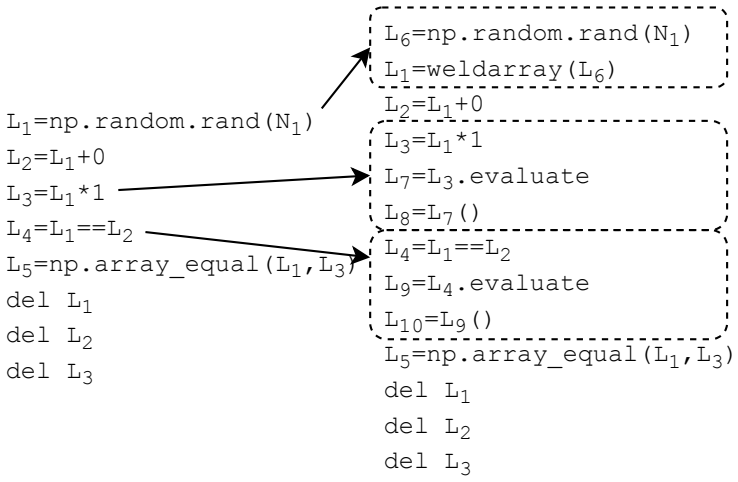


Figure 10 Translating NumPy to WeldNumpy.

WeldNumpy [30] was developed as a replacement for NumPy with better performance, which was achieved via two main techniques. First, WeldNumpy exploits lazy evaluation instead of eager evaluation, which is used in NumPy. Second, WeldNumpy implements its APIs using Weld IR [22], an intermediate representation designed for parallel data processing. Through lazy evaluation, the IR fragments of invoked APIs are combined into an IR program. During a true evaluation, the IR program is compiled and optimized for native hardware. Some major optimization techniques are loop fusion, loop tiling, and vectorization. WeldNumpy provides `weldarray`, a subclass of NumPy's `ndarray`. Thus, after an `ndarray` is converted to a `weldarray`, the new object supports most NumPy operations and enjoys improved performance.

However, as WeldNumpy is lazily evaluated, it requires users to explicitly call `evaluate()` when necessary. The `evaluate()` method should not be invoked too often; otherwise, the WeldNumpy runtime misses optimization opportunities and introduces overheads of compiling the Weld IR. Neither should it be too late as that would cause errors. Thus, a NumPy-to-WeldNumpy translator needs to figure out the appropriate positions to insert `evaluate()`.

The evaluating positions can be located by identifying *exposed lazy variables*. A variable is exposed if it is used beyond the DSL's APIs, which means the true value of the variable may be required, or it is alive at the end of the collected lazy IR, which allows potential external usage of the variable during subsequent execution. When a variable is exposed but lazy, it should be explicitly evaluated. Such variables can be identified within an one-pass scan of the lazy IR. The translator can thence insert `evaluate()` for these variables.

Figure 10 shows an example of translating NumPy to WeldNumpy. The translated IR first converts L_1, an `ndarray`, to a `weldarray`, such that L_2, L_3, and L_4 enjoy WeldNumpy's optimization. However, `np.array_equal()` is not supported by WeldNumpy; thus, operand L_3 has to be evaluated before being passed. While L_4 is explicitly evaluated because of potential exposure, L_2 remains lazy, since it is deleted and has no external use.

Such a translator leverages the laziness analysis enabled by Cunctator. It might be tempting to think that the translation could be done through a compiler without Cunctator. Note that that compiler would have to face the laziness analysis problem as Cunctator tackles; if it ignores that, its replacement of an eagerly evaluated NumPy API with a lazy evaluated WeldNumpy could cause errors. Doing the laziness analysis is difficult for a compiler for the many challenges (e.g., Python complexities, API-host interplay) mentioned in the introduction section.

6.4 From Pandas to Spark

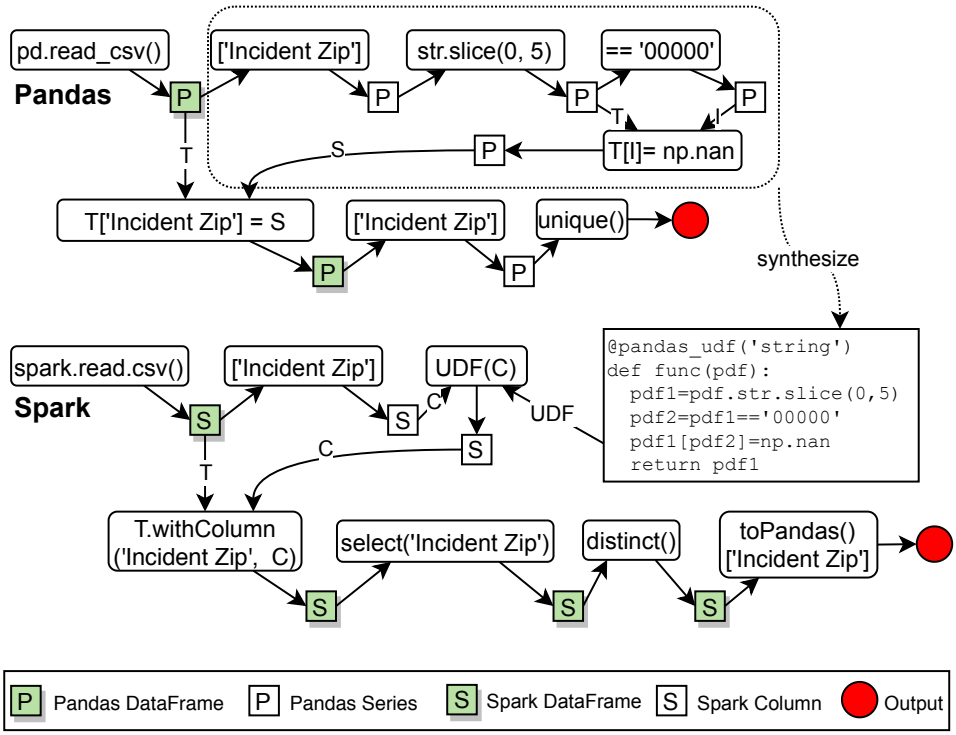


Figure 11 Pandas to Spark.

Both Pandas and Spark provide a class called `DataFrame`. They both represent logical tables, which have named and typed columns. While Pandas' operations in `DataFrame` are eagerly evaluated, most of Spark's `DataFrame` methods are lazily evaluated. During a true evaluation, Spark employs a code generation technique [20] to compile an operation sequence. Such technique renders the Spark DataFrame API a performant replacement of Pandas. In addition, Spark has native support for Pandas, including Pandas UDF [24], by which a user can apply Pandas operations to a Spark `Column`. Spark also contains type casting APIs that convert between Spark `DataFrame` and Pandas `DataFrame`. These features offers conveniences to translation of a Pandas program to a Spark program.

Similar to NumPy to WeldNumpy, the laziness analysis by Cunctator puts down the basis for the development of an automatic Pandas-to-Spark translator. Our prototype focuses on a common use pattern of Pandas: A program first loads a file as a `DataFrame`, then performs some operations on it, and finally outputs the result. In such a pattern, only one `DataFrame` object is involved, and no Pandas `DataFrame` object or `Series` (typically represents a column) object is exposed, so all instances of the two types only participate in Pandas operations. When such a pattern is matched, the translator tries to optimize it.

During the translation, the Pandas file loading function is replaced by a counterpart in Spark, thus creating a Spark `DataFrame`. Correspondingly, the `Series` objects selected from Pandas' `DataFrame` become Spark `Column` objects. If there is a sequence of operations on a `Series` that outputs another `Series`, the sequence is synthesized into a Pandas UDF for Spark, which is applied to the corresponding Spark `Column`. If a `Series` is assigned to the Pandas `DataFrame`, the corresponding `Column` is assigned to the Spark `DataFrame` as well. When an operation on a `Series` returns an object other than a `Series`, if the operation (e.g., `unique()`) has a counterpart in Spark, the `Column` is applied to the corresponding Spark operation, and then the result is converted to the expected type; otherwise (e.g., `diff()`), the `Column` is selected and converted to a `Series` before applying the operation. Figure 11 illustrates the translation for a Pandas program collected from the Pandas Cookbook [23].

7 API Redirection

If a DSL's runtime needs to leverage Cunctator to perform optimization, the optimizer developer needs to redirect the APIs in the DSL through renaming and rewriting. With the Cunctator framework, the process is made simple. For example, to redirect `numpy.add` in NumPy's runtime, current implementation of `numpy.add` could be renamed to `numpy._add`; then, a new implementation of `numpy.add` will just record API calls as lazy IR instructions and returns a lazy object as shown in Figure 6.

To simplify the process, Cunctator offers some utilities. For the aforementioned example, what the optimizer developer needs to write to put the following into the module *numpy*:

```
def add(*args, **kwargs):
   return lazy_call("numpy.add", numpy._add, args, kwargs,
       kwargsToUpdate={"out"})
```

Method `lazy_call` is the utility interface that Cunctator offers. Its first argument is for the annotation field of a `call` instruction (see §5). The argument `kwargsToUpdate` specifies that `numpy._add` is going to update only its argument `out` (if there is one). The call to `lazy_call` in this example will essentially materialize the method shown in Figure 6.

8 Efforts in Applying Cunctator

There is some work needed from the library developers. This work needs to be done only once for a given library; the results can benefit all programs using that library. This one-time work includes: (1) redirecting some APIs that are important for performance (other APIs can be left alone, which will be treated in the same way as host Python code is); (2) supporting MIN-watch for some common types; and (3) implementing optimization passes. Table 1 shows our prototype optimizers' summary in this work. For a common programmer that uses a library, the only change she needs to make to her code is to insert one or several lines of code to import the optimizer.

We initially considered automatic library transformations, but found that it was difficult to do for the complexities of Python. It is, for instance, often impossible for static code analysis to tell whether an argument is subject to modifications, due to dynamic types, aliases, higher-level functions, and inter-procedural complexities. The design choice made in Cunctator is a choice for practicability.

Table 1 Summary of optimizers.

Optimizer	#APIs*	Supported types†	Opt pass LoC‡
NumPy	45	`ndarray`, `dtype`	50 (§6.1) 93 (§6.3)
Spark	24	`RDD`, `StorageLevel`	201 (§6.2)
Pandas	28	`DataFrame`, `Series`	436 (§6.4)

* The number of redirected APIs.

† The types that support dependent discovery.

‡ Lines of code for implementing the optimization passes described in §6.

9 Evaluation

In this section, we conduct a series of experiments to (1) demonstrate the usefulness of the four optimizations (§6) enabled by Cunctator, and (2) measure the runtime overhead of Cunctator. Time usage is collected by the *timeit* command of jupyter [12], which adaptively chooses a number of repetitions in favor of timing accuracy. Peak memory usage is collected by *memit* command extended by *memory-profiler* [18], which profiles a program's memory usage line by line. The test platform for NumPy and Pandas is a Linux machine with Intel Xeon Silver 4114 CPUs. Spark programs run on a cluster of eight Linux machines with AMD Opteron 6128 CPUs.

9.1 Optimizers

We collect 15 programs for the experiments that are relevant to the example optimizations described in the previous section; five for each of the three packages (NumPy, Spark, Pandas). Thirteen of them were collected from GitHub; the other two were the examples used in the earlier sections of this paper – we included them to show the performance benefits for the described optimizations. Table 2 shows the descriptions and inputs of all benchmarks. Their source code can be found in the Docker image of Cunctator [5]. Figure 12 shows the speedups in different optimizer settings. Detailed results are presented in Table 3. Each program set is discussed separately in following subsections.

9.1.1 NumPy

The temporary variable reducer (abbr. *reduceTmp*) accelerates all benchmarks, with speedups ranging from 1.19X to 1.54X. The highest speedup is achieved on P1, because its operations are easy to compute and hence the cost of temporary variable is prominent. Besides time benefits, *reduceTmp* also reduces peak memory usage. P4 highlights the reduction with a rate of 75%. The high rate is because of pipelined operations, which means each temporary variable is only used one time and then discarded.

Table 2 Descriptions of collected benchmarks.

	Description	Input
	NumPy	
P1	Program in Figure 1a	vectors of size 10^9
P2	Compute vibration energy	vectors of size 5×10^8
P3	Find least-squares solution	vectors of size 5×10^8
P4	Find log-likelihood of $\mathcal{N}(\mu, \sigma^2)$	vectors of size 5×10^8
P5	Compute Black-Scholes model	vectors of size 10^8
	Spark	
P1	Program in Listing 1	text file of 90MB
P2	Demultiplex a file to multiple files	xml file of 244MB
P3	Transform data format	json file of 62MB
P4	Intersect IDs in two tables	two csv files of 34MB
P5	Find counts of different words	text file of 460MB
	Pandas	
P1	Find names of median occurrence	csv file of 273MB
P2	Find top complaints	csv file of 526MB
P3	Find ratios of noise complaints	csv file of 526MB
P4	Find unique zip codes after data cleaning	csv file of 526MB
P5	Find top occupations wrt. male ratio	csv file of 240MB

Table 3 Benchmark results.

	P1	P2	P3	P4	P5
NumPy time usage (mean ± std. dev.)					
baseline	12.5s±2.95ms	41.1s±75.4ms	27.4s±5.51ms	43.2s±57.6ms	39.1s±23ms
reduceTmp	8.14s±2.14ms	34s±15.3ms	22.9s±105ms	33.7s±26.9ms	32.9s±22.7ms
Weld 1T	6.38s±34.1ms	39.1s±87.6ms	21.8s±45ms	42.7s±144ms	22.6s±28.3ms
Weld 10T	995ms±9.24ms	27s±142ms	17.4s±60.3ms	15.6s±44.4ms	9.94s±35ms
NumPy peak memory usage (MB)					
baseline	38181	19131	19130	15316	9213
reduceTmp	30577	11503	15317	3873	6251
Spark time usage (mean ± std. dev.)					
baseline	31.9s±123ms	82s±698ms	38.5s±116ms	20.9s±36ms	49.1s±272ms
w/o opt	32.1s±215ms	81s±639ms	38.5s±178ms	21.1s±75.8ms	49.2s±392ms
optimized	17.1s±93.4ms	48.8s±348ms	28.8s±66.8ms	20.3s±317ms	47s±226ms
Pandas time usage (mean ± std. dev.)					
baseline	6.03s±50.3ms	9.65s±21.8ms	9.8s±13.4ms	9.72s±40.7ms	7.7s±37.4ms
Spark 1T	17.1s±100ms	5.51s±152ms	3.01s±127ms	5.76s±101ms	7.45s±241ms
Spark 10T	2.92s±203ms	951ms±51ms	690ms±26.9ms	1.3s±145ms	1.29s±59ms

For WeldNumpy converter, we test it with one thread (abbr. *Weld 1T*) and ten threads (abbr. *Weld 10T*) separately. *Weld 1T* shows speedups ranging from 1.01X to 1.95X. Because WeldNumpy currently supports only a limited number of NumPy APIs, for unsupported APIs, it needs to transform data from Weld format to NumPy format to perform the operations, and if necessary, the results need to be transformed back. As WeldNumpy evolves to support more APIs, *Weld 1T* is going to perform better. Moreover, with ten threads, WeldNumpy achieves significant speedups up to 12.5X. Note that Weld has built-in support for multi-threading but NumPy does not.

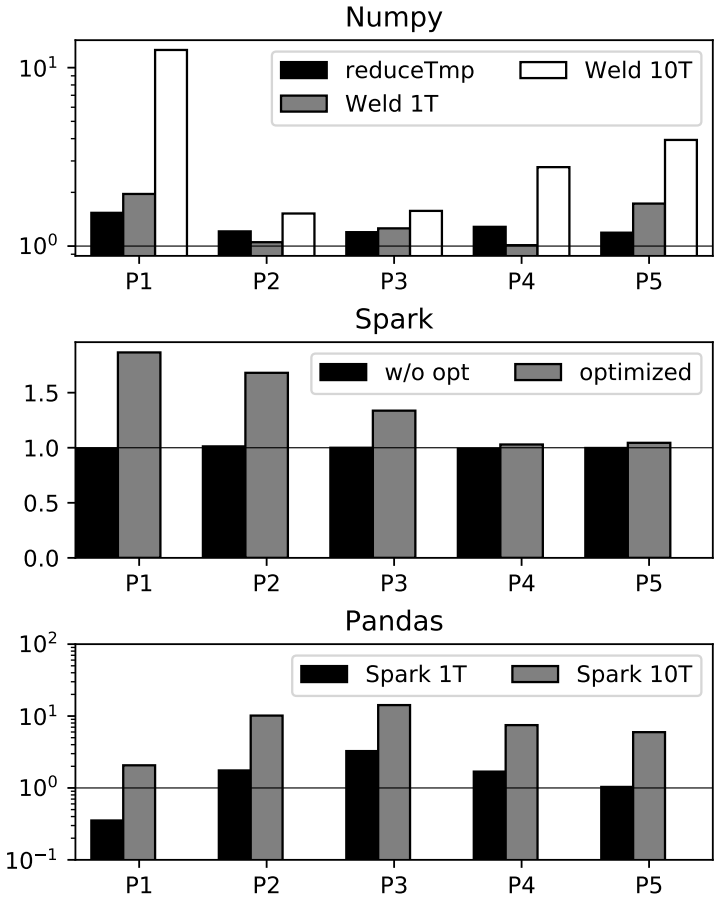


Figure 12 Speedups.

9.1.2 Spark

The Spark optimizer shows speedups ranging from 1.03X to 1.87X. Among the benchmarks, P1 and P2 lack `cache()`; P3 and P5 have unnecessary `cache()`; P4 has a `cache()` operation at a useless location, while the place that needs `cache()` does not have one. Our optimizer fixes them all. It adds `cache()` to P1 and P2, removes `cache()` from P3 and P5, and corrects P4 by removing the unnecessary `cache()` and adding one at the appropriate place.

In addition, we test the benchmarks with Cunctator enabled but optimizing pass disabled(abbr. `w/o opt`). The results show no performance degradation. This confirms that MIN-watch has almost no overhead for non-watched objects, as PySpark programs typically invoke user defined functions written in Python frequently.

9.1.3 Pandas

The Pandas-to-Spark optimizer is tested with one Spark thread (abbr. `Spark 1T`) and ten Spark threads (abbr. `Spark 10T`). Note that Spark supports multi-threading but Pandas does not. `Spark 1T` shows speedups on three programs. This is impressive because, while Pandas enjoys the high performance of SIMD instructions, Spark's query compiler emits Java bytecode. The slowdown on P1 is dominated by Spark's CSV loader, which performs much worse than Pandas' loader in this case. Nevertheless, `Spark 10T` enjoys speedups as high as 14.2X.

Table 4 Overhead (percentage of 10s program runs).

		CPS				
		50	500	1000	2000	10000
Threshold	50	0	0.85	0.85	1.7	0
	500	0.35	3.05	1.05	1.8	0
	1000	**0.25**	**2.35**	**2.25**	**1.7**	**0**
	2000	0.15	2.15	2.05	2.5	0.1
	10000	0.15	1.25	1.75	2.9	11.8

9.2 Overheads

For programs that cannot be optimized, a major concern is the overhead, which is highly related to the number of lazy IR instructions recorded. To investigate the overhead in different cases, we design an adversarial case for stress-testing:

```
def cps_simulator(M, N):
   for i in range(M):
      numpy.ones(N)
```

The program calls M times of `numpy.ones(N)`, which initializes a vector of size N. By tuning M and N, we can control the number of calls per second (CPS) and the total run time. We then combine some representative values of CPS and overhead control thresholds (see §4.3). For each combination, we run a ten-second experiment with Cunctator. By subtracting the results with the corresponding baseline results, we obtain an overhead matrix, shown as Table 4.

The overhead increases when CPS increases. When CPS exceeds the threshold, Cunctator disables itself for the later part of the run; the overhead drops. For the default threshold (1000), the worst overhead is 2.35%, which happens in the extreme case where there are 1000 function calls per second. In practice, a program is unlikely to have a stable CPS rate close to the threshold, thus the overhead is much lower. In addition, it is worth noting that Cunctator is mainly implemented in Python, except for the MIN-watch. If we reimplement some critical components in C, such as the lazy IR evaluator, a lower overhead is expected.

It is worth noting that, in the domains that we explored, the number of relatives per object is few, hence our benchmarks bear little overhead of finding relatives. For example, a NumPy array usually has no relative if its buffer belongs to itself, or only one relative if its buffer is from another object. For domains where deeply nested objects are common, the overhead control threshold can be adjusted to fit the need of the domains.

9.3 Threats to Validity

Cunctator is evaluated based on Python 3.7.3, NumPy 1.17.0, WeldNumpy 0.0.1, Pandas 0.25.0, and Spark 2.4.3. The APIs and implementation of these software packages may change after new versions are released. Thus the new releases may invalidate our optimization techniques and evaluation results. Nevertheless, new patterns of API misuses related to these new releases are likely to appear. Unless the new versions employ a technique similar to BELE, Cunctator can be leveraged to optimize the new patterns.

The soundness of a Cunctator-based optimizer relies on the correctness of the API knowledge provided by the optimizer's developer. Such knowledge includes how to discover the relatives of an object, which arguments of an API could be updated during the API call, and how to apply optimization passes onto the recorded lazy IR. If any of the knowledge is incorrect, programs optimized by Cunctator may yield unexpected results.

10 Related Work

Lazy evaluation has been studied extensively in functional programming [9, 11, 3, 16, 10]. Scala [21] provides a `lazy` keyword to express the call-by-need semantics of a variable. However, Scala does not manage the potential side-effect of a thunk, the expression bound to the lazy variable; thus, the correctness of lazy evaluation relies on the programmer. Many hosted DSLs (e.g., Spark [31] and TensorFlow [1]) employ lazy evaluation; their limitations have been discussed in §1.

There are some studies on optimizing DSLs. Weld [22] and its limitations have been discussed and compared with. Delite [26] is a framework for developing Scala-hosted DSLs by leveraging generative programming [6]. Similarly to Cunctator, it lazily evaluates DSL operations and logs them as a form of IR, which will be optimized and executed at a certain point in time. However, Delite provides no mechanism to handle the dependencies between DSL operations and their host code.

There are several earlier studies (e.g., telescoping languages [13], Broadway [8]) that try to use manual annotations of libraries to help optimizations. They give no systematic considerations of the host-API dynamic dependencies. Numba [15] is a JIT compiler of Python that targets optimizing manipulations of `ndarray` in NumPy. AutoGraph [19] employs static code conversion and generative programming to transform PyTorch-style programs to TensorFlow-style programs. All these methods and tools offer a closed set of optimization techniques for specific program semantics. Cunctator does not include any optimization technique but provides a general framework to simplify the creation of a DSL optimizer.

Finally, the NumPy optimizer presented in Section 6.1 replaces list copies with in-place updates. In this sense, it is similar to deforestation, an optimization technique usually used in programming environments where referential transparency ends up being very costly[29, 14, 7, 27].

11 Conclusion

This paper introduces the concept of BELE, and describes MIN-watch, the first efficient runtime monitoring method tailored to data dependence analysis between host code and APIs for BELE. The paper demonstrates the usefulness of Cunctator in enabling four optimizations that are not supported by existing frameworks, giving 1.03-14.2X speedups. While Cunctator targets Python-hosted DSLs, we believe the potentially applicability of the techniques goes much beyond Python.

References

1 Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283. USENIX Association, 2016. URL: `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi`.

2 Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

3 Adrienne G. Bloss, Paul Hudak, and Jonathan Young. Code optimizations for lazy evaluation. *LISP Symb. Comput.*, 1(2):147–164, 1988.

4 Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981. doi:10.1007/BFb0025774.

5 Cunctator Docker Image. https://github.com/sangongs/Cunctator_docker.

6 Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming - methods, tools and applications*. Addison-Wesley, 2000. URL: http://www.addison-wesley.de/main/main.asp?page=englisch/bookdetails&productid=99258.

7 Bruno Morais Ferreira, Britaldo Silveira Soares-Filho, and Fernando Magno Quintão Pereira. The dinamica EGO virtual machine. *Sci. Comput. Program.*, 173:3–20, 2019. doi:10.1016/j.scico.2018.02.002.

8 Samuel Z. Guyer and Calvin Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proc. IEEE*, 93(2):342–357, 2005. doi:10.1109/JPROC.2004.840489.

9 Peter Henderson and James H. Morris Jr. A lazy evaluator. In Susan L. Graham, Robert M. Graham, Michael A. Harrison, William I. Grosky, and Jeffrey D. Ullman, editors, *Conference Record of the Third ACM Symposium on Principles of Programming Languages, Atlanta, Georgia, USA, January 1976*, pages 95–103. ACM Press, 1976. doi:10.1145/800168.811543.

10 Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In Barbara G. Ryder and Brent Hailpern, editors, *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, pages 1–55. ACM, 2007. doi:10.1145/1238844.1238856.

11 Thomas Johnsson. Efficient compilation of lazy evaluation. In Mary S. Van Deusen and Susan L. Graham, editors, *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, Montreal, Canada, June 17-22, 1984*, pages 58–69. ACM, 1984. doi:10.1145/502874.502880.

12 Project Jupyter. https://jupyter.org/.

13 Ken Kennedy, Bradley Broom, Arun Chauhan, Robert J. Fowler, John Garvin, Charles Koelbel, Cheryl McCosh, and John M. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proc. IEEE*, 93(2):387–408, 2005. doi:10.1109/JPROC.2004.840447.

14 Georgios Korfiatis, Michalis A. Papakyriakou, and Nikolaos Papaspyrou. A type and effect system for implementing functional arrays with destructive updates. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Federated Conference on Computer Science and Information Systems - FedCSIS 2011, Szczecin, Poland, 18-21 September 2011, Proceedings*, pages 879–886, 2011. URL: http://ieeexplore.ieee.org/document/6078196/.

15 Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a llvm-based python JIT compiler. In Hal Finkel, editor, *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, November 15, 2015*, pages 7:1–7:6. ACM, 2015. doi:10.1145/2833157.2833162.

16 John Launchbury. A natural semantics for lazy evaluation. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 144–154. ACM Press, 1993. doi:10.1145/158511.158618.

17 Wes McKinney. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, 14, 2011.

18 Python memory-profiler Library. https://pypi.org/project/memory-profiler/.

19 Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. Autograph: Imperative-style coding with graph-based performance. *CoRR*, abs/1810.08061, 2018. arXiv:1810.08061.

20 Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, 2011. doi:10.14778/2002938.2002940.

21 Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with scala. *Commun. ACM*, 57(4):76–86, 2014. doi:10.1145/2591013.

22 Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

23 Pandas Cookbook Example. https://nbviewer.jupyter.org/github/jvns/pandas-cookbook/tree/v0.1/cookbook/.

24 Introducing Pandas UDF for PySpark. https://databricks.com/blog/2017/10/30/introducing-vectorized-udfs-for-pyspark.html.

25 Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Tensors and dynamic neural networks in python with strong gpu acceleration, 2017.

26 Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, 2014. doi:10.1145/2584665.

27 Sebastian Ullrich and Leonardo de Moura. Counting immutable beans: Reference counting optimized for purely functional programming. *CoRR*, abs/1908.05647, 2019. arXiv:1908.05647.

28 Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Comput. Sci. Eng.*, 13(2):22–30, 2011. doi:10.1109/MCSE.2011.37.

29 Philip Wadler. Deforestation: Transforming programs to eliminate trees. In Harald Ganzinger, editor, *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer, 1988. doi:10.1007/3-540-19027-9_23.

30 WeldNumpy. https://www.weld.rs/weldnumpy/.

31 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In Erich M. Nahum and Dongyan Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010. URL: https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets.

Accelerating Object-Sensitive Pointer Analysis by Exploiting Object Containment and Reachability

Dongjie He
University of New South Wales, Sydney, Australia

Jingbo Lu
University of New South Wales, Sydney, Australia

Yaoqing Gao
Huawei, Toronto, Canada

Jingling Xue
University of New South Wales, Sydney, Australia

Abstract

Object-sensitive pointer analysis for an object-oriented program can be accelerated if context-sensitivity can be selectively applied to some precision-critical variables/objects in the program. Existing pre-analyses, which are performed to make such selections, either preserve precision but achieve limited speedups by reasoning about all the possible value flows in the program conservatively or achieve greater speedups but sacrifice precision (often unduly) by examining only some but not all the value flows in the program heuristically. In this paper, we introduce a new approach, named Turner, that represents a sweet spot between the two existing ones, as it is designed to enable object-sensitive pointer analysis to run significantly faster than the former approach and achieve significantly better precision than the latter approach. Turner is simple, lightweight yet effective due to two novel aspects in its design. First, we exploit a key observation that some precision-uncritical objects can be approximated based on the object-containment relationship pre-established (by applying Andersen's analysis). This approximation introduces a small degree yet the only source of imprecision into Turner. Second, leveraging this initial approximation, we introduce a simple DFA to reason about object reachability for a method intra-procedurally from its entry to its exit along all the possible value flows established by its statements to finalize its precision-critical variables/objects identified. We have validated Turner with an implementation in Soot against the state of the art using a set of 12 popular Java benchmarks and applications.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases Object-Sensitive Pointer Analysis, CFL Reachability, Object Containment

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.16

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.12`

Funding This work is supported by an ARC DP grant (DP180104069) and a UNSW-Huawei research grant (YBN2019105002).

Acknowledgements We thank the reviewers for their constructive comments.

1 Introduction

Pointer analysis is a significant static program analysis that approximates the potential runtime values (memory locations) for the pointer variables in a program. It plays an important role in a wide range of real-world applications, including security analysis [2, 10], program verification [8], program understanding [36, 20], and bug detection [25, 11].

For object-oriented languages like Java, *context sensitivity*, which distinguishes the variables declared and objects allocated locally in a method under different calling contexts, is widely enforced in developing highly precise pointer analyses. In general, a context is represented by a sequence of k context elements (under k limiting). There are two common

forms of context-sensitivity: k-call-site-sensitivity [29] (which distinguishes the contexts of a method by its k-most-recent call sites) and k-object-sensitivity [23] (which distinguishes the contexts of a method by its receiver object's k-most-recent allocation sites). The latter is widely regarded as a better abstraction in achieving precision and efficiency [31, 39, 41, 12, 22].

However, k-object-sensitive pointer analysis (with k-object-sensitivity as its context abstraction), denoted $k$obj, still does not scale well for reasonably large programs when $k \geqslant 3$ and is often time-consuming when it is scalable [31, 39, 41, 12]. As k increases, blindly applying a k-limiting context abstraction uniformly to a program can cause the number of contexts handled to blow up exponentially (often without improving precision much).

In this paper, we address the problem of developing a pre-analysis for a Java program to enable $k$obj to apply context-sensitivity (i.e, a k-limited context abstraction) only to some of its variables/objects selected and context-insensitivity to all the rest in the program.

▶ **Definition 1.** *A variable/object n in a program is* precision-critical *if $k$obj loses precision in terms of the points-to information obtained (for some value of k) when n is analyzed by $k$obj context-insensitively instead of context-sensitively.*

A pre-analysis is said to be *precision-preserving* if it can identify the precision-critical variables/objects in a program precisely or over-approximately as being context-sensitive, and *non-precision-preserving* otherwise. Unfortunately, making such selections precisely is out of question as solving $k$obj without k-limiting is undecidable [27]. When designing a practical pre-analysis, which aims to select the set of context-sensitive variables/objects, C_{ideal}, in the program, the main challenge are to ensure that (1) C_{ideal} includes as many precision-critical variables/objects as possible but as few precision-uncritical variables/objects as possible, (2) C_{ideal} results in no or little precision loss, and (3) C_{ideal} is found in a lightweight manner to ensure that the pre-analysis overhead introduced is negligible (relative to $k$obj).

Recently, several pre-analyses have been proposed [32, 13, 9, 19, 22, 21]. Broadly speaking, two approaches exist. Eagle [22, 21] represents a precision-preserving acceleration of $k$obj by reasoning about CFL (Context-Free-Language) reachability in the program. Designed to be precision-preserving, Eagle analyzes conservatively and often efficiently the value flows reaching a variable/object and selects the set of context-sensitive variables/objects as a superset of the set of precision-critical variables/objects in the program over-approximately, thereby limiting the potential speedups achieved. On the other hand, Zipper [19], as a non-precision-preserving representative of the remaining pre-analyses [32, 13, 9, 19], examines the value flows reaching a variable/object heuristically and often efficiently by selecting the set of context-sensitive variables/objects to include some but not all the precision-critical variables/objects and also some precision-uncritical variables/objects in the program. As a result, Zipper can sometimes improve the efficiency of $k$obj more significantly than Eagle but at the expense of introducing a substantial loss of precision for some programs.

In this paper, we introduce a new approach, named Turner, that represents a sweet spot between Eagle and Zipper: Turner enables $k$obj to run significantly faster than Eagle while achieving significantly better precision than Zipper. Despite losing a small precision in the average points-to set size (#avg-pts), Turner achieves exactly the same precision for the other three commonly used precision metrics [31, 39, 41, 12, 22, 21], call graph construction (#call-edges), may-fail casting (#may-fail-casts) and polymorphic call detection (#poly-calls), for a set of 12 popular Java benchmarks and applications evaluated. Turner is simple, lightweight yet effective due to two novel aspects in its design. First, we exploit a key observation that some precision-uncritical objects can be approximated initially based on the object-containment relationship that is inferred from the points-to information pre-computed by Andersen's analysis [1]. This approximation turns out to be practically

accurate, as it introduces a small degree yet the only source of imprecision into the final points-to information computed. Second, leveraging this initial approximation, we introduce a simple DFA (Deterministic Finite Automaton) to reason about object reachability across a method (from its entry to its exit) intra-procedurally along all the possible value flows established by its statements to finalize all its precision-critical variables/objects selected.

We have validated Turner with an implementation in Soot against Eagle and Zipper using a set of 12 Java benchmarks and applications. In general, Turner enables $k$obj to run significantly faster than Eagle due to fewer precision-uncritical variables/objects analyzed context-sensitively and achieve significantly better precision than Zipper due to more precision-critical variables/objects analyzed context-sensitively than Zipper.

In summary, our paper makes the following contributions:

- We introduce a new approach, Turner, that can accelerate k-object-sensitive pointer analysis (i.e., $k$obj) for Java programs significantly with negligible precision loss.
- We propose to first approximate the precision-criticality of the objects in a program based on object containment and then decide whether the variables/objects in the program should be context-sensitive or not by conducting an object reachability analysis intra-procedurally with a DFA, which turns out to be simple, lightweight and effective.
- Turner enables $k$obj to run significantly faster than Eagle and achieve significantly better precision than Zipper for a set of 12 popular Java benchmarks and applications evaluated in terms of four common precision metrics, #avg-pts, #call-edges, #may-fail-casts, and #poly-calls (with Turner losing no precision for the last three metrics).

The rest of this paper is organized as follows. Section 2 motivates our Turner approach. Section 3 gives a version of $k$obj that supports selective context-sensitivity. Section 4 formalizes our Turner approach. In Section 5, we evaluate Turner against the state of the art. Section 6 discusses the related work. Finally, Section 7 concludes the paper.

2 Motivation

We motivate Turner in the context of the two state-of-the-art pre-analyses, Eagle [22, 21] and Zipper [19]. Eagle supports partial context-sensitivity as it enables $k$obj to analyze only a subset of variables/objects in a method context-sensitively. On the other hand, Zipper allows $k$obj to analyze a method either fully context-sensitively or fully context-insensitively. Like Eagle, Turner also supports partial context-sensitivity in order to maximize the potential speedups attainable. As in Eagle and Zipper, Turner also relies on the points-to information in a program pre-computed by Andersen's analysis [1] (context-insensitively).

In Section 2.1, we give some background information. In Section 2.2, we examine the main challenges faced in developing a pre-analysis for accelerating $k$obj and discuss the methodological differences between Turner and two existing approaches, Eagle and Zipper. In Section 2.3, we introduce a motivating example abstracted from real code by highlighting the effects of these differences on the context-sensitivity selectively applied to $k$obj. In Section 2.4, we describe the basic idea behind Turner (including our insights and trade-offs).

2.1 Background

In object-sensitive pointer analysis [23], the calling contexts of a method are distinguished by its receiver objects. Let each allocation site be abstracted by one unique object. In $k$obj, an object o_1 is modeled context-sensitively by a *heap context* of length $k-1$, $[o_2, ..., o_k]$, where o_i is the receiver object of a method in which o_{i-1} is allocated. As a result, a method with

o_1 as its receiver object will be analyzed context-sensitively multiple times, once for each of o_1's heap contexts $[o_2, ..., o_k]$, i.e., once under every possible *method context* $[o_1, ..., o_k]$ of length k. Thus, kOBJ can be specified by either heap or method contexts alone.

Given a variable v analyzed under a method context c, its context-sensitive points-to set is expressed as $\mathsf{pts}(v, c) = \{(o_1, c_1), \cdots, (o_n, c_n)\}$, where each pointed-to object o_i is identified by its heap context c_i. Let M_v be the set of method contexts under which v is analyzed. Then the context-insensitive points-to set for v can be found as $\overline{\mathsf{pts}}(v) = \bigcup_{c \in M_v} \{o \mid (o, c') \in \mathsf{pts}(v, c)\}$. When comparing different context-sensitive pointer analyses precision-wise, the context-insensitive points-to information thus obtained is used, as is done in the literature [32, 12, 13, 39, 19, 21].

2.2 Challenges

A variable/object n in a program is precision-critical if kOBJ loses precision when it analyzes n context-insensitively instead of context-sensitively (Definition 1). In the case of a precision loss, there must exist some variable v in the program such that its context-insensitive points-to information becomes less precise. In this case, $\overline{\mathsf{pts}}(v)$ will contain not only the pointed-to objects found before (when n is analyzed context-sensitively) but also some spurious pointed-to objects introduced now (when n is analyzed context-insensitively). As n and v can be further apart in the program, separated by a long sequence of method calls (with complex field accesses on n along the way), designing a practical pre-analysis P, which selects a set of variables/objects in a program for kOBJ to analyze context-sensitively, is challenging (since solving kOBJ without k-limiting is undecidable [27]). For a program, let C_{ideal} be the set of precision-critical variables/objects specified by Definition 1 and C_P the set of context-sensitive variables/objects selected by P. The main challenges lie in how to ensure that (1) $|C_{\text{ideal}} - C_P|$ is minimized (so that as many precision-critical variables/objects are selected) and $|C_P - C_{\text{ideal}}|$ is minimized (so that as few precision-uncritical variables/objects are selected), (2) C_P causes kOBJ to lose no or little precision, and (3) C_P is selected in a lightweight manner (so that P introduces negligible overhead relative to kOBJ).

A pre-analysis for kOBJ relies on the following fact to identify a precision-critical variable/object, with its accesses possibly triggered by statements outside its containing method. Without loss of generality, a method is assumed to contain only one return statement of the form "**return** r", where r a local variable in the method (referred to as its *return variable*).

▶ **Fact 2.** *Consider a program being analyzed object-sensitively with the parameters and the return variable of a method modeled as its (special) fields as in [22, 21]. A variable/object n in a method M in the program is considered to be precision-critical* only if*, during program execution, there is a value flow entering and leaving M via a parameter or the return variable of M, by passing through n (i.e., by first writing into n via an access path and then reading it from the same access path), where n may be the parameter or the return variable itself.*

In this case, analyzing n context-sensitively will allow several such value flows to be tracked separately based on their calling contexts. Otherwise, some precision may be potentially lost.

A pre-analysis, as illustrated in Figure 1, should identify a (local) variable `x` as precision-critical by considering a total of four possible value-flow patterns passing through `x` (classified according to whether the two end points of a value-flow are a parameter or the return variable of its containing method [34, 22]). The same four patterns are also applicable to a locally allocated object. In "*param-return*" (Figure 1(a)), the pre-analysis should recognize that the object created in line 8 will flow into `x` in `id()` via its parameter `p` and then out of `id()` via a return variable, which happens to be `x` itself. In "*return-param*" (Figure 1(b)), the

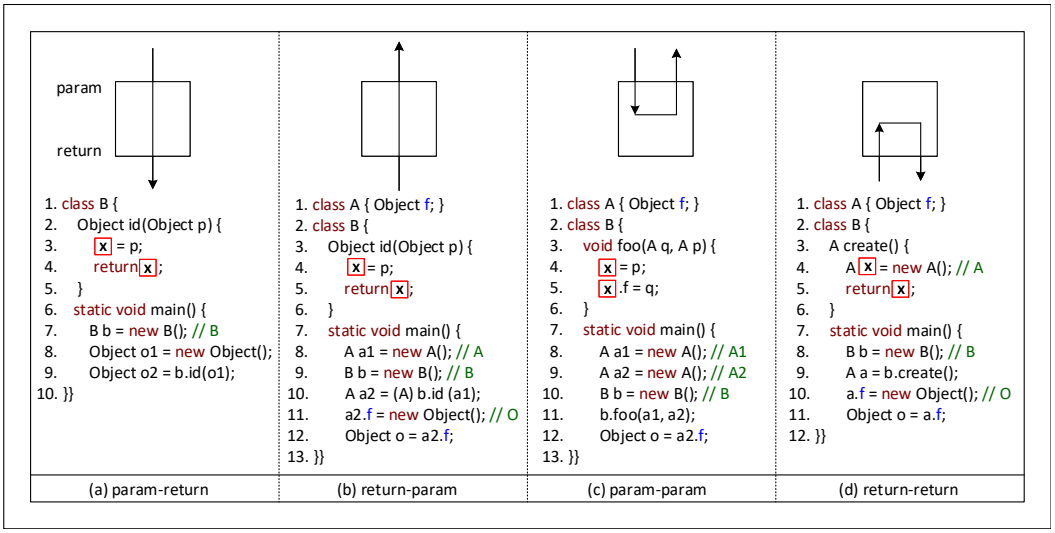


Figure 1 A total of four possible value-flow patterns for determinng whether a variable `x` should be precision-critical or not.

pre-analysis, when checking whether the object created in line 11 will flow into `o` in line 12 or not, will first need to find out what `a2` points to. This will entail reasoning about the value flow of `a2` in reverse order, by entering `id()` via its return statement (variable) and leaving `id()` from its parameter `p`. In "*param-param*" (Figure 1(c)), the object `A1` created in line 8 will flow into `x` (or `x.f` precisely) in `foo()` via its parameter `q` and then out of `foo()` via its parameter `p`. In "*return-return*" (Figure 1(d)), the pre-analysis, when checking whether the object created in line 10 can flow into `o` in line 11 or not, will need to find what `a` points to, by entering and exiting `create()` from its return variable and visiting `x` in between.

We can now discuss how TURNER differs from EAGLE [22, 21] and ZIPPER [19] methodologically. To start with, all the three are relatively lightweight with respect to kOBJ. Below we examine these pre-analyses in terms of their efficiency and precision tradeoffs made on approximating C_{ideal}. There are two caveats. First, C_{ideal} is conceptual but cannot be found exactly in a program. Second, some precision-critical variables/objects affect the precision and/or efficiency of kOBJ more profoundly than others, but they cannot be easily identified. How to do so approximately can be an interesting research topic in future work.

EAGLE is precision-preserving, since it accounts for all the four value-flow patterns given in Figure 1 by reasoning about CFL reachability in the program inter-procedurally to ensure that $C_{\text{ideal}} - C_{\text{EAGLE}} = \emptyset$. For some programs, EAGLE may conservatively misclassify many precision-uncritical variables/objects as being precision-critical, thereby causing $C_{\text{EAGLE}} - C_{\text{ideal}}$ to be unduly large, and consequently, limiting the speedups attainable.

ZIPPER is not precision-preserving (implying that $C_{\text{ideal}} - C_{\text{ZIPPER}} \neq \emptyset$, in general), since it considers only the "*param-return*" and "*return-param*" patterns in Figure 1 heuristically by pattern-matching and ignores "*param-param*" (according to its authors [19]) and "*return-return*" (according to its open-source implementation). For some programs, ZIPPER can achieve greater speedups than EAGLE (under certain configurations that dictate how certain objects should be analyzed) but at a precision loss, since it has misclassified some precision-yet performance-critical variables/objects as context-insensitive.

In this paper, TURNER is designed to strike a good balance between EAGLE and ZIPPER. We aim to ensure that $|C_{\text{TURNER}} - C_{\text{ideal}}| < |C_{\text{EAGLE}} - C_{\text{ideal}}|$ so that TURNER can enable kOBJ to run significantly faster than EAGLE (due to fewer precision-uncritical variable/objects selected

```
1. class Entry {
2.   Object key, value;
3.   Entry(Object p, Object q) {
4.     this.key = p;
5.     this.value = q;
6. }}

7. class HashMap {
8.   Entry[] table;
9.   Object get(Object k){
10.    int idx = k.hashCode;
11.    Entry[] t = this.table;
12.    Entry e = t[idx];
13.    Object r = e.value;
14.    return r;
15.  }
16.  void put(Object k, Object v) {
17.    int idx = k.hashCode;
18.    Entry e = new Entry(k, v);  // E
19.    Entry[] t = this.table;
20.    t[idx] = e;
21.  }
22.  HashMap() {
23.    Entry[] t = new Entry[16]; // @
24.    this.table = t;
25. }}

26. class A {
27.   void foo(Object k) {
28.     HashMap map1 = new HashMap(); // M1
29.     HashMap map2 = new HashMap(); // M2
30.     Object v1 = new Object(); // O1
31.     Object v2 = new Object(); // O2
32.     map1.put(k, v1);
33.     map2.put(k, v2);
34.     Object w1 = map1.get(k);
35.     Object w2 = map2.get(k);
36.  }
37.  public static void main(String args[]) {
38.    Object k = new Object(); // O
39.    A ai = new A(); // Ai        1 ≤ i ≤ n
40.    ai.foo(k);
41.    ...
42. }}
```

Figure 2 A Java program abstracted from real code using the standard JDK library.

for kOBJ to analyze context-sensitively). At the same time, we aim to ensure that $|C_{\text{ideal}} - C_{\text{TURNER}}| < |C_{\text{ideal}} - C_{\text{ZIPPER}}|$ so that TURNER can also enable kOBJ to achieve significantly better precision than ZIPPER (due to more precision-critical variable/objects selected for kOBJ to analyze context-sensitively). We accomplish this by exploiting object containment to approximate the precision-criticality of objects and then reasoning about object reachability by considering all the four value-flow patterns in Figure 1 intra-procedurally.

2.3 Example

Figure 2 gives a Java program abstracted from real code developed based on JDK. In lines 1-25, a simplified `HashMap` class is defined. In lines 26-42, class `A` represents a use case of `HashMap`. In `foo()`, two instances of `HashMap`, `M1` and `M2`, and two instances of `java.lang.Object`, `O1` and `O2`, are created. Afterwards, `O1` (`O2`), pointed to by `v1` (`v2`), is deposited into `M1` (`M2`), pointed to by `map1` (`map2`), with `O` (received from its parameter `k`) as the corresponding key, and later retrieved and saved in `w1` (`w2`). In `main()`, n instances of `A`, $\texttt{A}_1, ..., \texttt{A}_n$, are created (where $n > 1$) and then used as the receivers for invoking `foo()`.

Table 1 lists the contexts used for analyzing this program by the four main analyses, 2OBJ, E-2OBJ, Z-2OBJ, and T-2OBJ. Here, P-2OBJ denotes the version of 2OBJ that adopts the selective context-sensitivity prescribed by $P \in \{\text{E (for EAGLE)}, \text{Z (for ZIPPER)}, \text{T (for TURNER)}\}$. EAGLE is always precision-preserving. For this program, ZIPPER happens to be also precision-preserving since Z-2OBJ behaves exactly as 2OBJ does. TURNER also happens to be precision-preserving but T-2OBJ differs from 2OBJ/Z-2OBJ and E-2OBJ substantially. Below we focus on examining how the context-insensitive points-to information for `w1` and `w2` in `foo()`, $\overline{\texttt{pts}}(\texttt{w1}) = \{\texttt{O1}\}$ and $\overline{\texttt{pts}}(\texttt{w2}) = \{\texttt{O2}\}$, is obtained by each of the four main analyses. For reasons of symmetry, Figure 3 illustrates only how $\overline{\texttt{pts}}(\texttt{w1}) = \{\texttt{O1}\}$ is obtained.

Table 1 The contexts used for analyzing the variables/objects in Figure 2 by 2obj, E-2obj, Z-2obj, and T-2obj (where i in each context containing $\texttt{A}_i/\texttt{a}_i$ ranges over $[1, n]$).

Method	Variables/Objects	2obj/ Z-2obj	E-2obj	T-2obj
Entry	p, q, this	[E, M1], [E, M2]	[E, M1], [E, M2]	[E, M1], [E, M2]
get	k	[M1, A_i], [M2, A_i]	[]	[]
	e, r, this, t		[M1, A_i], [M2, A_i]	[M1], [M2]
put	k, v, e, this, t	[M1, A_i], [M2, A_i]	[M1, A_i], [M2, A_i]	[M1], [M2]
	E	[M1], [M2]	[M1], [M2]	
HashMap	this, t	[M1, A_i], [M2, A_i]	[M1, A_i], [M2, A_i]	[M1], [M2]
	@	[M1], [M2]	[M1], [M2]	
foo	v1, v2, w1, w2	[A_i]	[]	[]
	O1, O2			
	k, map1, map2		[A_i]	
	M1, M2			
main	k, a_i	[]	[]	[]
	O, A_i			

First of all, 2obj analyzes `foo()` for a total of n times by identifying its variables/objects under the i-th invocation with its receiver $\texttt{A}_i$ (Figure 3(a)). Thus, $\forall\ 1 \leqslant i \leqslant n : \texttt{pts}(\texttt{w1}, [\texttt{A}_i]) = \{\texttt{O1}, [\texttt{A}_i]\} \wedge \texttt{pts}(\texttt{w2}, [\texttt{A}_i]) = \{\texttt{O2}, [\texttt{A}_i]\}$ context-sensitively. By projecting out all the contexts, 2obj obtains $\overline{\texttt{pts}}(\texttt{w1}) = \{\texttt{O1}\}$ and $\overline{\texttt{pts}}(\texttt{w2}) = \{\texttt{O2}\}$ context-insensitively, as desired.

For this particular program, Z-2obj is equivalent to 2obj (Table 1 and Figure 3(a)). However, it is easy to modify it slightly so that Z-2obj will behave differently while suffering from a loss of precision (as it does not consider the last two patterns given in Figure 1).

E-2obj enables 2obj to support partial context-sensitivity without losing any precision. The variables/objects in $\{\texttt{v1}, \texttt{v2}, \texttt{w1}, \texttt{w2}, \texttt{O1}, \texttt{O2}\}$ in `foo()` and variable `k` in `get()` will now be context-insensitive. In the case of `foo()`, however, `k`, `map1`, `map2`, `M1` and `M2` must still be analyzed context-sensitively due to a spurious "*param-return*" pattern established by the facts that (1) `k` is a parameter, (2) `put()` can write into `M1`/`M2`, and (3) `get()` can read from `M1`/`M2`. As a result, as illustrated in Figure 3(b), E-2obj will still need to analyze `foo()` for a total of n times, since it must distinguish the two `HashMap` objects `M1` and `M2` created in `foo()` context-sensitively as in 2obj, except that it can now analyze the two objects, `O1` and `O2`, created in `foo()` context-insensitively. Thus, E-2obj obtains directly that $\texttt{pts}(\texttt{w1}, [\,]) = \{\texttt{O1}, [\,]\}$ and $\texttt{pts}(\texttt{w2}, [\,]) = \{\texttt{O2}, [\,]\}$, i.e., $\overline{\texttt{pts}}(\texttt{w1}) = \{\texttt{O1}\}$ and $\overline{\texttt{pts}}(\texttt{w2}) = \{\texttt{O2}\}$.

T-2obj, as illustrated in Figure 3(c), goes beyond E-2obj (for this particular program) by modeling `M1` and `M2` also context-insensitively. As a result, `foo()` is analyzed context-insensitively only once. As in the case of E-2obj, T-2obj also obtains directly that $\texttt{pts}(\texttt{w1}, [\,]) = \{\texttt{O1}, [\,]\}$ and $\texttt{pts}(\texttt{w2}, [\,]) = \{\texttt{O2}, [\,]\}$, i.e., $\overline{\texttt{pts}}(\texttt{w1}) = \{\texttt{O1}\}$ and $\overline{\texttt{pts}}(\texttt{w2}) = \{\texttt{O2}\}$.

2.4 Our Approach

Turner is designed to accelerate $k$obj with partial context-sensitivity at a negligible loss of precision. Unlike Eagle [22, 21] and Zipper [19], Turner works by exploiting both object containment and object reachability to enable $k$obj to strike a better balance between efficiency and precision. In principle, Turner may lose precision in its first stage only but will always preserve precision in its second stage if it does not lose precision in its first stage.

2.4.1 Object Containment

To start with, we exploit a key insight stated below to identify some precision-uncritical objects approximately based on the object containment relationship that is inferred from the points-to information pre-computed (context-insensitively) by Anderson's analysis [1].

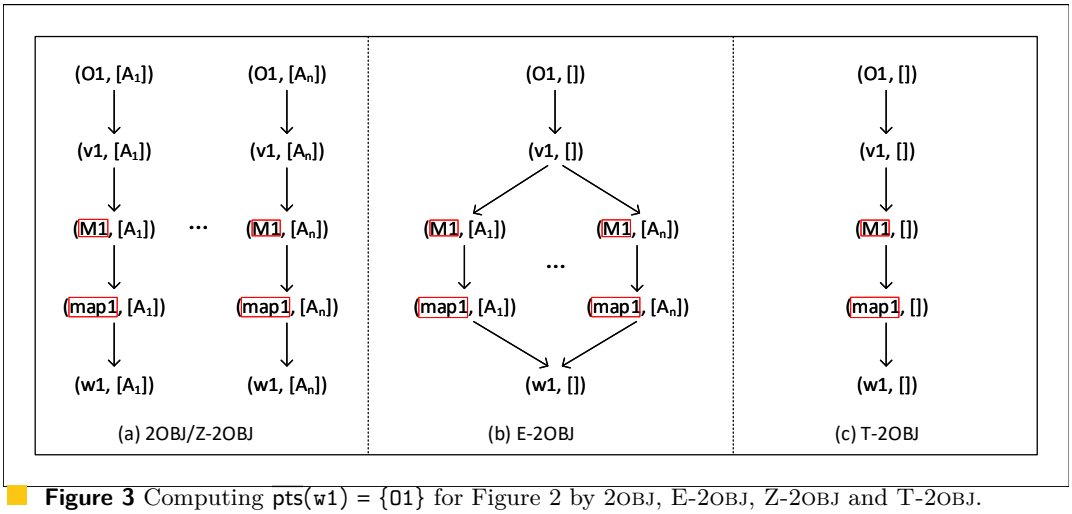


Figure 3 Computing pts(w1) = {O1} for Figure 2 by 2OBJ, E-2OBJ, Z-2OBJ and T-2OBJ.

▶ **Observation 3.** *A* top container *is an object that is pointed to by neither (1) another object (which may be the container itself) via a field of a declared type of C or C[], where C is a class type nor (2) the return variable of the method in which the container is allocated.*

A bottom container *is an object that does not point to another object (which may be the container itself) via a field of a declared type of C or C[], where C is a class type.*

Given a program, its top and bottom containers are considered as being precision-uncritical.

▶ **Definition 4.** *Observation 3 is said to be precision-preserving for a program if k*OBJ *does not lose precision when it analyzes the precision-uncritical objects identified in the program context-insensitively and the remaining variables/objects exactly as before.*

Therefore, an object created by a factory method (regarded here as a method that returns its own allocated objects via its return variable) is not a top container. Such an object will be considered as being precision-uncritical iff it is a bottom container. For a program, the precision-uncritical objects identified here will be analyzed by *k*OBJ context-insensitively (for the reasons given below) and the remaining objects will be further classified as either precision-critical or precision-uncritical by an object reachability analysis (Section 2.4.2).

Consider `create()` in Figure 1(d). The object `A` created inside is not regarded as a top container, since it is pointed to by its return variable. In object-sensitive pointer analysis, when `create()` called on receiver object `B` in line 9 is analyzed, returning `A` to this caller is actually modeled as `this.ret = x` (line 5) and `a = b.ret` (line 9), where both `this` and `b` point to `B`, and `ret` can be understood as a special return variable introduced for `create()` (Section 4.2.2.2) [22, 21]. Conceptually, `A` is not a top container. In this example, `A` is not a bottom container either, since `A.f = O` in line 10, where `O` is an instance of `java.lang.Object`. As a result, `A` is considered as being precision-critical. However, if lines 10-11 were not present, then `A` would be deemed as being precision-uncritical as it is now a bottom container.

Consider Figure 2 (which is free of factory methods), where a total of $n + 7$ objects can be found: `E`, `C`, `M1`, `M2`, `O1`, `O2`, `O`, $\mathtt{A}_1, ..., \mathtt{A}_n$. According to the object containment relationship inferred from Andersen's analysis, `M1` and `M2` contain `C`, which contains `E`, which contains `O1`, `O2` and `O`. By Observation 3, the set of top containers is given by $\{\mathtt{M1}, \mathtt{M2}, \mathtt{A}_1, ...\mathtt{A}_n\}$ and the set of bottom containers is given by $\{\mathtt{O1}, \mathtt{O2}, \mathtt{O}, \mathtt{A}_1, ...\mathtt{A}_n\}$. Note that both sets of containers are not necessarily disjoint. Thus, the $n + 5$ objects in $\{\mathtt{M1}, \mathtt{M2}, \mathtt{O1}, \mathtt{O2}, \mathtt{O}, \mathtt{A}_1, ...\mathtt{A}_n\}$ are considered as being precision-uncritical and will thus be analyzed by *k*OBJ context-insensitively.

In our approach, Observation 3 (made based on object containment) represents the only source of imprecision in Turner, which may propagate into its object reachability analysis. Turner will suffer only a slight loss of precision in #avg-pts computed by T-$k$obj when some top or bottom containers that should be context-sensitive are mis-classified as being precision-uncritical, and consequently, analyzed by T-$k$obj context-insensitively. However, this does not affect the precision of #call-edges, #may-fail-casts, and #poly-calls for the set of 12 popular Java programs evaluated (at least). The set of top containers consists of the objects that are allocated and used locally in a method, such as `M1` and `M2` (two `HashMap` objects) in `foo()` in Figure 2. These objects do not require context-sensitivity, since their encapsulated data does not usually flow out of its containing methods via their parameters or return variables. On the other hand, a bottom container also does not usually require context-sensitivity, as it represents an object that typically encapsulates its primitive data (if any), including arrays of primitive types if it ever contains pointers, such as `O`, `O1` and `O2` (three field-less `java.lang.Object` objects) in Figure 2. In Section 5.3, we will examine two examples to explain why Turner loses some small precision in #avg-pts but preserves precision in #call-edges, #may-fail-casts, and #poly-calls in real code.

2.4.2 Object Reachability

Given a program, Turner relies on a simple DFA to reason about implicitly the four value-flow patterns in Figure 1 in a method to select its variables/objects to be analyzed by T-$k$obj context-sensitively. By design, the precision-uncritical objects identified by Observation 3 in the program are deemed context-insensitive. The remaining objects in the program will be classified by the DFA as either precision-critical (context-sensitive) or precision-uncritical (context-insensitive). Simultaneously, the variables in the program are classified. Turner's intra-procedural analysis will be precision-preserving if Observation 3 is precision-preserving, as it is designed to over-approximate the precision-critical variables/objects in the program.

For our example in Figure 2, Table 1 gives the contexts selected by Turner for $k$obj, i.e., T-2obj. We discuss only their differences with the contexts selected by Eagle for $k$obj, i.e., E-2obj. By exploiting object containment as discussed in Section 2.4.1, `M1`, `M2`, `O1`, `O2`, and `O` have been identified as being precision-uncritical and will thus be analyzed context-insensitively. Given that `M1` and `M2`, are now context-insensitive, `k`, `map1`, and `map2` will also be identified as being context-insensitive by our DFA, as the spurious "*param-param*" pattern that causes Eagle to flag `M1`, `M2`, `k`, `map1`, and `map2` in `foo()` as being context-sensitive no longer exists (Section 2.3). As `M1` and `M2` are context-insensitive, the contexts $[\texttt{M1}, \texttt{A}_i]$ and $[\texttt{M2}, \texttt{A}_i]$ listed under E-2obj have been shortened to `[M1]` and `[M2]` under T-2obj (Table 1).

3 Preliminaries

We take a standard formalization of $k$obj [23] from [35] and adapt it to support selective context-sensitivity. This gives a formal basis to understand our pre-analysis introduced.

3.1 A Simplified Object-Oriented Language

We consider a simplified object-oriented language, i.e., a subset of Java, in which a program consists of a set of classes, where each class consists of static/instance fields and methods. Table 2 gives six kinds of statements, which are labeled by their line numbers, in the language operated on by $k$obj. Note that "x = **new** T(...)" in Java is modeled as "x = **new** T; x.⟨`init`⟩(...)", where ⟨`init`⟩`()` is the corresponding constructor invoked. Section 5 discusses how to handle other complex language features such as reflection and native code.

Table 2 Six types of statements analyzed by kOBJ.

Kind	Statement	Description
new	$l : v = \textbf{new}\ T$	v is a local variable and T is a class type
assign	$l : v = v'$	v and v' are local variables
assignglobal	$l : v = v'$	v or v' is a global variable
load	$l : v = v'.f$	v and v' are local variables and f is a field name
store	$l : v.f = v'$	v and v' are local variables and f is a field name
call	$l : b = a_0.m(a_1, ..., a_r)$	b and a_i are local variables and m is an instance method

As kOBJ is context-sensitive but flow-insensitive, the control flow statements in a program are irrelevant. As is standard with several recent implementations of kOBJ [31, 39, 41, 12], static fields are analyzed context-insensitively as global variables, but static methods can be analyzed context-sensitively as instance methods as follows. For a static method $m()$ defined in class C, a call to $m()$ can be interpreted as $this.m()$ by proceeding as if $m()$ were an instance method defined in `java.lang.Object` and inherited by C. Given $this.m()$, $m()$ can then be analyzed context-sensitively under the receiver object pointed to by $this$, which is the receiver object of m's closest (instance) caller method, if any, on the call stack.

Finally, every method is assumed to have one single return statement of the form "**return** r", where r is a local variable (referred to as its return variable). Note that a return statement in a method is not listed explicitly in Table 2, as it will be handled implicitly at a call statement where the method is invoked (as shown in Figure 4).

3.2 Selective Object-Sensitive Pointer Analysis

Given a program, let $\mathbb{M}$, $\mathbb{F}$, $\mathbb{H}$, $\mathbb{V}$, $\mathbb{G}$ and $\mathbb{L}$ be its sets of methods, fields, allocation sites, local variables, global variables, and statements (identified by their labels, e.g., line numbers), respectively.

Let $\mathbb{C} = \mathbb{H}^*$ be the universe of contexts. Given a context $ctx = [o_1, ..., o_n] \in \mathbb{C}$ and a context element o, we write $o +\!\!+ ctx$ for $[o, o_1, ..., o_n]$ and $\lceil ctx \rceil_k$ for $[o_1, ..., o_k]$.

The rules used for performing kOBJ will make use of the following functions:

- $\mathsf{methodOf} : \mathbb{L} \mapsto \mathbb{M}$
- $\mathsf{methodCtx} : \mathbb{M} \mapsto \wp(\mathbb{C})$
- $\mathsf{dispatch} : \mathbb{M} \times \mathbb{H} \mapsto \mathbb{M}$
- $\mathsf{len} : \mathbb{V} \cup \mathbb{G} \cup \mathbb{H} \mapsto \mathbb{N}$
- $\mathsf{pts} : (\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \times \mathbb{C} \mapsto \wp(\mathbb{H} \times \mathbb{C})$

where methodOf gives the containing method of a statement, methodCtx keeps track of the (method) contexts used for analyzing a method, dispatch resolves a virtual call to its target method, len defines the length of contexts used for analyzing a variable/object, and pts records the points-to information found for a variable or an object's field.

Figure 4 gives five rules used by kOBJ for analyzing six kinds of statements in Table 2 with two kinds of assignments processed together in one rule. In [NEW], v points to the object o_l uniquely identified by its allocation site l. Note that $\lceil ctx \rceil_{\mathsf{len}(o_l)}$ is the heap context of o_l (Section 2.1). In [ASSIGN/ASSIGNGLOBAL], two kinds of assignments, where v and v' are either local or global variables, are handled as copies. In [STORE] and [LOAD], field accesses are analyzed in the standard manner. In [CALL], a call to an instance method $b = a_0.m(a_1, ..., a_r)$ is analyzed. We write $this^{m'}$, $p_i^{m'}$ and $ret^{m'}$ for the "this" variable, i-th parameter and return variable of m', respectively, where m' is a target method resolved. Frequently, we also write

$$\frac{l : v = \text{new } T \quad M = \text{methodOf}(l) \quad ctx \in \text{methodCtx}(M)}{(o_l, \lceil ctx \rceil_{\text{len}(o_l)}) \in \text{pts}(v, \lceil ctx \rceil_{\text{len}(v)})} \quad \text{[New]}$$

$$\frac{l : v = v' \quad M = \text{methodOf}(l) \quad ctx \in \text{methodCtx}(M)}{\text{pts}(v', \lceil ctx \rceil_{\text{len}(v')}) \subseteq \text{pts}(v, \lceil ctx \rceil_{\text{len}(v)})} \quad \text{[Assign/AssignGlobal]}$$

$$\frac{l : v.f = v' \quad M = \text{methodOf}(l) \quad ctx \in \text{methodCtx}(M) \quad (o, hctx) \in \text{pts}(v, \lceil ctx \rceil_{\text{len}(v)})}{\text{pts}(v', \lceil ctx \rceil_{\text{len}(v')}) \subseteq \text{pts}(o.f, hctx)} \quad \text{[Store]}$$

$$\frac{l : v = v'.f \quad M = \text{methodOf}(l) \quad ctx \in \text{methodCtx}(M) \quad (o, hctx) \in \text{pts}(v', \lceil ctx \rceil_{\text{len}(v')})}{\text{pts}(o.f, hctx) \subseteq \text{pts}(v, \lceil ctx \rceil_{\text{len}(v)})} \quad \text{[Load]}$$

$$\frac{\begin{array}{c} l : b = a_0.m(a_1, ..., a_r) \quad M = \text{methodOf}(l) \quad ctx \in \text{methodCtx}(M) \\ (o, hctx) \in \text{pts}(a_0, \lceil ctx \rceil_{\text{len}(a_0)}) \quad m' = \text{dispatch}(m, o) \quad ctx' = o +\!\!+ hctx \end{array}}{\begin{array}{c} ctx' \in \text{methodCtx}(m') \quad (o, hctx) \in \text{pts}(this^{m'}, \lceil ctx' \rceil_{\text{len}(this^{m'})}) \\ \forall i \in [1, r] : \text{pts}(a_i, \lceil ctx \rceil_{\text{len}(a_i)}) \subseteq \text{pts}(p_i^{m'}, \lceil ctx' \rceil_{\text{len}(p_i^{m'})}) \quad \text{pts}(ret^{m'}, \lceil ctx' \rceil_{\text{len}(ret^{m'})}) \subseteq \text{pts}(b, \lceil ctx \rceil_{\text{len}(b)}) \end{array}} \quad \text{[Call]}$$

Figure 4 Rules for kOBJ formalized to support selective context-sensitivity.

$p_0^{m'}$ for $this^{m'}$. In the conclusion of this rule, $ctx' \in \text{methodCtx}(m')$ reveals how the method contexts of a method are introduced. Initially, $\text{methodCtx}(\text{“}\texttt{main}\text{”}) = \{[\,]\}$.

kOBJ represents a k-object-sensitive pointer analysis with a $(k-1)$-context-sensitive heap (by handling global variables context-insensitively as is standard) [31, 39, 41, 12]. Thus, kOBJ selects the context lengths for different entities e in $\mathbb{V} \cup \mathbb{G} \cup \mathbb{H}$ differently as follows:

$$\text{len}_{k\text{OBJ}}(e) = \begin{cases} 0 & e \in \mathbb{G} \\ k & e \in \mathbb{V} \\ k-1 & e \in \mathbb{H} \end{cases} \tag{1}$$

As a pre-analysis, Turner will select a subset $CI_{\text{Turner}} \subseteq \mathbb{V} \cup \mathbb{H}$ so that kOBJ will analyze CI_{Turner} context-insensitively but $(\mathbb{V} \cup \mathbb{H}) \setminus CI_{\text{Turner}}$ context-sensitively as follows:

$$\text{len}_{\text{Turner}}(e) = \begin{cases} 0 & e \in CI_{\text{Turner}} \\ \text{len}_{k\text{OBJ}}(e) & e \in (\mathbb{V} \cup \mathbb{G} \cup \mathbb{H}) \setminus CI_{\text{Turner}} \end{cases} \tag{2}$$

As discussed earlier, Eagle [22] will also enable kOBJ to analyze only a subset of variables/objects in a method context-sensitively but Zipper [19] will require a method (i.e., all its variables/objects) to be analyzed either fully context-sensitively or fully context-insensitively.

4 Turner: Our Approach

We describe the two stages of Turner, object containment (Section 4.1) and reachability (Section 4.2), by focusing predominantly on formalizing our object reachability analysis.

4.1 Object Containment

In this first stage on object containment analysis, we identify some precision-uncritical objects in a program based on the points-to information pre-computed by Andersen's analysis [1] according to Observation 3. For an object o, we write ret_o to denote the return variable in the method where o is allocated. For two objects o_1 and o_2, we write $o_1 \xrightarrow{class-type(f)} o_2$ if $o_1.f = o_2$ for some field f whose declared type is either C or $C[\,]$, where C is some class type.

As a result, the set of precision-uncritical objects in a program can be found by:

$$\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}} = \mathsf{TopCon} \cup \mathsf{BotCon} \tag{3}$$

where the sets of top and bottom containers in the program are identified as follows:

$$\begin{aligned} \mathsf{TopCon} &= \left\{ o \;\middle|\; \left(\nexists\, (o', f) \in \mathbb{H} \times \mathbb{F} : o' \xrightarrow{class\text{-}type(f)} o \right) \wedge ret_o \text{ does not point to } o \right\} \\ \mathsf{BotCon} &= \left\{ o \;\middle|\; \nexists\, (o', f) \in \mathbb{H} \times \mathbb{F} : o \xrightarrow{class\text{-}type(f)} o' \right\} \end{aligned} \tag{4}$$

4.2 Object Reachability

In this second stage on object reachability analysis, we make use of a DFA to determine intra-procedurally whether a variable/object requires context-sensitivity or not. Let CI_{TURNER} be the set of context-insensitive variables/objects that are finally selected by TURNER to support selective context-sensitivity required in (2). By design, $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}} \subseteq CI_{\mathrm{TURNER}}$, i.e., the precision-uncritical objects selected earlier will always be analyzed context-insensitively. The remaining objects and all the variables in the program will be further classified as either context-sensitive or context-insensitive according to the DFA, by leveraging $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}}$.

We first review a standard formulation for performing pointer analysis intra-procedurally based on CFL (Context-Free Language) reachability (Section 4.2.1). We then evolve it incrementally into a DFA-based intra-procedural reachability analysis (Section 4.2.2).

4.2.1 Standard CFL-Reachability-based Pointer Analysis

A parameterless method that contains no calls inside can be represented by a directed graph G, known as PAG (Pointer Assignment Graph), with its nodes drawn from $\mathbb{V} \cup \mathbb{G} \cup \mathbb{H}$ and its five types of edges added according to the rules given in Figure 5 [37, 28]. Loads and stores to the elements of an array are modeled by collapsing all the elements into a special field `arr` of the array. For each PAG edge $x \xrightarrow{\ell} y$ with its label ℓ, its inverse edge is denoted as $y \xrightarrow{\overline{\ell}} x$.

$$\frac{l : v = \textbf{new}\ T}{o_l \xrightarrow{\mathsf{new}} v \quad v \xrightarrow{\overline{\mathsf{new}}} o_l}\ [\text{P-NEW}] \qquad \frac{v = v'.f}{v' \xrightarrow{\mathsf{load[f]}} v \quad v \xrightarrow{\overline{\mathsf{load[f]}}} v'}\ [\text{P-LOAD}] \qquad \frac{v.f = v'}{v' \xrightarrow{\mathsf{store[f]}} v \quad v \xrightarrow{\overline{\mathsf{store[f]}}} v'}\ [\text{P-STORE}]$$

$$\frac{v = v'}{v' \xrightarrow{\mathsf{assign}} v \quad v \xrightarrow{\overline{\mathsf{assign}}} v'}\ [\text{P-ASSIGN}] \qquad \frac{v = v'}{v' \xrightarrow{\mathsf{assignglobal}} v \quad v \xrightarrow{\overline{\mathsf{assignglobal}}} v'}\ [\text{P-ASSIGNGLOBAL}]$$

Figure 5 Rules for creating the PAG edges for a method containing no calls inside.

Let L be a CFL over Σ formed by the edge labels in G. Each path p in G has a string $L(p)$ in Σ^* formed by concatenating in order the labels of edges in p. A node v in G is *L-reachable* from a node u in G if there exists a path p from u to v, known as *L-path* and denoted by $L(u, v)$, such that $L(p) \in L$. For a node n in G, we write $L(u, v)^n$ if n appears on $L(u, v)$. For a path p in G such that its label is $L(p) = \ell_1, \cdots, \ell_r$ in L, the inverse of p, i.e., $\overline{p}$ has the label $L(\overline{p}) = \overline{\ell_r}, \cdots, \overline{\ell_1}$.

We start with a standard grammar that defines the following language L_0 [37, 28]:

$$L_0 : \begin{cases} \mathit{flowsto} \longrightarrow \mathsf{new}\ \mathit{flows}^* \\ \mathit{flows} \longrightarrow \mathsf{assign} \mid \mathsf{assignglobal} \mid \mathsf{store[f]}\ \mathit{alias}\ \mathsf{load[f]} \\ \overline{\mathit{flowsto}} \longrightarrow \overline{\mathit{flows}}^*\ \overline{\mathsf{new}} \\ \overline{\mathit{flows}} \longrightarrow \overline{\mathsf{assign}} \mid \overline{\mathsf{assignglobal}} \mid \overline{\mathsf{load[f]}}\ \mathit{alias}\ \overline{\mathsf{store[f]}} \\ \mathit{alias} \longrightarrow \overline{\mathit{flowsto}}\ \mathit{flowsto} \end{cases} \tag{5}$$

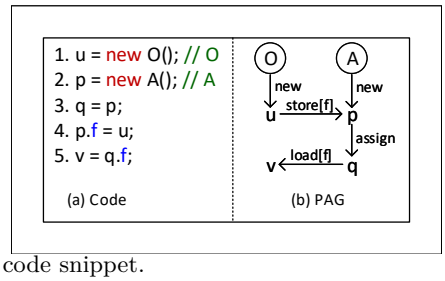


Figure 6 The PAG for a code snippet.

If o *flowsto* v, then v is L_0-reachable from o, i.e., $L_0(o, v)$. To handle aliases, $\overline{\mathit{flowsto}}$ is introduced as the inverse of the *flowsto* relation. A *flowsto* path p can be inverted to obtain its corresponding $\overline{\mathit{flowsto}}$ path $\overline{p}$ using its inverse edges, and vice versa. Thus, o *flowsto* x iff x $\overline{\mathit{flowsto}}$ o. This means that $\overline{\mathit{flowsto}}$ actually represents the standard points-to relation. As a result, x *alias* y iff x $\overline{\mathit{flowsto}}$ o *flowsto* y for some object o, so that field accesses are handled precisely by solving a *balanced parentheses* problem. For the code snippet (consisting of local variables only), together with its PAG, depicted in Figure 6, we know that $L_0(O, v)$, i.e., `O` *flowsto* `v`, implying that `v` points to `O`, which holds due to the following *flowsto* path:

$$\mathtt{O} \xrightarrow{\mathsf{new}} \mathtt{u} \xrightarrow{\mathsf{store[f]}} \mathtt{p} \xrightarrow{\overline{\mathsf{new}}} \mathtt{A} \xrightarrow{\mathsf{new}} \mathtt{p} \xrightarrow{\mathsf{assign}} \mathtt{q} \xrightarrow{\mathsf{load[f]}} \mathtt{v} \tag{6}$$

By inverting all the edges in this *flowsto* path, a $\overline{\mathit{flowsto}}$ path showing `v` $\overline{\mathit{flowsto}}$ `O` is obtained.

4.2.2 Turner's Context-Sensitivity-Deciding Reachability Analysis

We will now over-approximate L_0 incrementally to obtain a regular grammar, i.e., a DFA to decide intra-procedurally whether a variable/object requires context-sensitivity or not.

4.2.2.1 Ignoring Context-Insensitive Value Flows

Instead of computing points-to information in a program directly, TURNER is designed to analyze the context-sensitive value flows across the parameters or return variables of its methods (Fact 2). Thus, we will ignore the assignglobal statements and the precision-uncritical objects in $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}}$, as all the value-flows passing through them are context-insensitive.

$$\frac{l : v = \mathbf{new}\ T \quad o_l \notin \mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}}}{o_l \xrightarrow{\mathsf{cs\text{-}likely}} o_l}\ [\text{P-OBJECT}]$$

Figure 7 Rule for treating all the objects in $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}}$ as context-insensitive.

To handle the objects in $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}}$ context-insensitively as global variables, as shown in Figure 7, we have added a self-loop edge label, named cs-likely, for each object that is not in

$\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}}$ to indicate that it is currently treated as being potentially context-sensitive but will be classified later as being either context-sensitive or context-insensitive by our reachability analysis. By deleting the two terminals assignglobal and $\overline{\text{assignglobal}}$ from and adding one new terminal cs-likely to the grammar for defining L_0, we obtain:

$$L_1 : \begin{cases} \mathit{flowsto} \longrightarrow \mathsf{new}\ \mathit{flows}^* \\ \mathit{flows} \longrightarrow \mathsf{assign} \mid \mathsf{store[f]}\ \mathit{alias}\ \mathsf{load[f]} \\ \overline{\mathit{flowsto}} \longrightarrow \overline{\mathit{flows}}^*\ \overline{\mathsf{new}} \\ \overline{\mathit{flows}} \longrightarrow \overline{\mathsf{assign}} \mid \overline{\mathsf{load[f]}}\ \mathit{alias}\ \overline{\mathsf{store[f]}} \\ \mathit{alias} \longrightarrow \overline{\mathit{flowsto}}\ \mathsf{cs\text{-}likely}\ \mathit{flowsto} \end{cases} \tag{7}$$

We will discuss how to handle method parameters and method calls shortly below.

Let us consider Figure 6 again by making two independent changes to the code snippet:

- If q is a global variable, then $\mathtt{p} \xrightarrow{\mathsf{assign}} \mathtt{q}$ will become $\mathtt{p} \xrightarrow{\mathsf{assignglobal}} \mathtt{q}$. As a result, $L_1(\mathtt{O}, \mathtt{v})$ can no longer be established as in (6) earlier (due to the absence of assignglobal in L_1).
- if A is a cs-likely object, then $L_1(\mathtt{O}, \mathtt{v})$ can also be established as before, since we have:

$$\mathtt{O} \xrightarrow{\mathsf{new}} \mathtt{u} \xrightarrow{\mathsf{store[f]}} \mathtt{p} \xrightarrow{\overline{\mathsf{new}}} \mathtt{A} \xrightarrow{\mathsf{cs\text{-}likely}} \mathtt{A} \xrightarrow{\mathsf{new}} \mathtt{p} \xrightarrow{\mathsf{assign}} \mathtt{q} \xrightarrow{\mathsf{load[f]}} \mathtt{v} \tag{8}$$

Otherwise, $L_1(\mathtt{O}, \mathtt{v})$ will no longer be possible due to the absence of $\mathtt{A} \xrightarrow{\mathsf{cs\text{-}likely}} \mathtt{A}$.

To simplify matters, returning values from a method can be treated identically as passing parameters for the method. In object-sensitive pointer analysis [31, 39, 41, 12, 22, 21], a method M is analyzed context-sensitively under different receiver objects. Thus, its return statement "**return** r" can be modeled as "*this.ret* = r", where *ret* is a fresh local variable (interpreted now as the *return variable* of M) and the return values in "*this.ret*" can be retrieved by its callers via its receiver objects. Given this simple transformation, the four value-flow patterns given in Figure 1 can be unified as one "*param-param*" pattern.

▶ **Lemma 5.** *A variable/object n in a method M requires context-sensitivity only if n lies on a sequence of statements, $s_1, ..., s_r$, such that (1) s_i and s_{i+1} form a def-use chain involving only local variables and cs-likely objects, (2) s_1 represents a use of either a cs-likely object or a parameter of M, and (3) s_r represents a definition of $P.f$, where P is a parameter of M (including this) and f is a field of the objects pointed by P (including M's return variable (ret)).*

Proof. Follows directly from Fact 2 and the definition of cs-likely objects. ◀

In this case, n should be context-sensitive, since the modification effects of different definitions of n on $P.f$ under different calling contexts of M must be separated context-sensitively.

4.2.2.2 Approximating the Value Flows Spanning across Method Calls

We now consider how to handle a method call made in a method being analyzed. TURNER will over-approximate the context-sensitive value flows spanning across a call site without analyzing its invoked methods. With L_1, we can only reason about CFL reachability starting from an object. With L_2 given below, we can also start from a variable (Lemma 5):

$$L_2 : \begin{cases} \mathit{flows} \longrightarrow (\mathsf{new} \mid \mathsf{assign} \mid \mathsf{store[f]}\ \mathit{alias}\ \mathsf{load[f]})^* \\ \overline{\mathit{flows}} \longrightarrow (\overline{\mathsf{new}} \mid \overline{\mathsf{assign}} \mid \overline{\mathsf{load[f]}}\ \mathit{alias}\ \overline{\mathsf{store[f]}})^* \\ \mathit{alias} \longrightarrow \overline{\mathit{flows}}\ \mathsf{cs\text{-}likely}\ \mathit{flows} \end{cases} \tag{9}$$

▶ **Lemma 6.** *Let G be the PAG built by the rules in Figures 5 and 7. $L_2 \supseteq L_1$.*

Proof. Follows simply from examining the structural differences in their productions. ◀

In both languages, the aliases between two variables are established in exactly the same way.

Next, we over-approximate L_2 to obtain L_3 by abstracting the field accesses with 1-limited access paths and handling aliases more conservatively (as explained shortly below):

$$L_3 : \begin{cases} \mathit{flows} \longrightarrow (\mathsf{new} \mid \mathsf{assign} \mid \mathsf{load} \mid \mathsf{store}\ \mathit{alias})^* \\ \overline{\mathit{flows}} \longrightarrow (\overline{\mathsf{new}} \mid \overline{\mathsf{assign}} \mid \overline{\mathsf{load}} \mid \mathit{alias}\ \overline{\mathsf{store}})^* \\ \mathit{alias} \longrightarrow \overline{\mathit{flows}}\ \mathsf{cs\text{-}likely}\ \mathit{flows} \end{cases} \tag{10}$$

Thus, the fields in loads and stores are ignored, and loads and assignments become indistinguishable, but stores are treated differently (i.e., unsymmetrically as loads) in order to keep track of aliases as desired. Note that L_3 is is still a CFL, since (1) a store is required to match a $\overline{\mathsf{new}}$, $\overline{\mathsf{assign}}$ or $\overline{\mathsf{load}}$, and (2) a $\overline{\mathsf{store}}$ is required to match a new, assign or load. This balanced-parentheses property is somehow hidden in the *alias*-production.

For the code given in Figure 6, $L_3(\mathtt{O}, \mathtt{v})$ will still hold even if, say, `v = q.f` is replaced by `v = q.g` due to the existence of the following *flowsto* path:

$$\mathtt{O} \xrightarrow{\mathsf{new}} \mathtt{u} \xrightarrow{\mathsf{store}} \mathtt{p} \xrightarrow{\overline{\mathsf{new}}} \mathtt{A} \xrightarrow{\mathsf{cs\text{-}likely}} \mathtt{A} \xrightarrow{\mathsf{new}} \mathtt{p} \xrightarrow{\mathsf{assign}} \mathtt{q} \xrightarrow{\mathsf{load}} \mathtt{v} \tag{11}$$

▶ **Lemma 7.** *Let G be the PAG built by the rules in Figures 5 and 7. $L_3 \supseteq L_2$.*

Proof. In L_3, the first two productions can be expressed equivalently as $\mathit{flows} \longrightarrow (\mathsf{new} \mid \mathsf{assign} \mid \mathsf{load} \mid \mathsf{store}\ \mathit{alias}\ \mathsf{load?})^*$ and $\overline{\mathit{flows}} \longrightarrow (\overline{\mathsf{new}} \mid \overline{\mathsf{assign}} \mid \overline{\mathsf{load}} \mid \overline{\mathsf{load}?}\ \mathit{alias}\ \overline{\mathsf{store}})^*$. Here, (s)? indicates that s is optional, where '(' and ')' can be omitted if s represents one symbol. We can conclude that $L_3 \supseteq L_2$ by noting that the field access paths in L_3 are 1-limited. ◀

In L_3, a store can now also be matched with a $\overline{\mathsf{store}}$ when looking for aliases:

$$\mathit{flows} \Longrightarrow^+ \ldots\ \mathsf{store}\ \overline{\mathit{flows}}\ \mathsf{cs\text{-}likely}\ \mathit{flows}\ \overline{\mathsf{store}}\ \ldots \tag{12}$$

For the code given in Figure 6, $L_3(\mathtt{O}, \mathtt{v})$ will thus still hold if we (1) replace `v = q.f` by `q.g = v` and (2) add `v = new V()`, where the allocated object, `V`, is assumed to be cs-likely:

$$\mathtt{O} \xrightarrow{\mathsf{new}} \mathtt{u} \xrightarrow{\mathsf{store}} \mathtt{p} \xrightarrow{\overline{\mathsf{new}}} \mathtt{A} \xrightarrow{\mathsf{cs\text{-}likely}} \mathtt{A} \xrightarrow{\mathsf{new}} \mathtt{p} \xrightarrow{\mathsf{assign}} \mathtt{q} \xrightarrow{\overline{\mathsf{store}}} \mathtt{v} \xrightarrow{\overline{\mathsf{new}}} \mathtt{V} \xrightarrow{\mathsf{cs\text{-}likely}} \mathtt{V} \xrightarrow{\mathsf{new}} \mathtt{v} \tag{13}$$

We discuss below how to exploit this property to avoid analyzing the methods invoked at a call site while still keeping track of all context-sensitive value flows spanning the call site.

$$\frac{b \ = \ a_0.m(a_1, \ldots, a_r)}{\forall\ i : a_i \xrightarrow{\mathsf{store}[p_i^{m'}]} a_0 \quad \forall\ i : a_0 \xrightarrow{\overline{\mathsf{store}[p_i^{m'}]}} a_i \quad a_0 \xrightarrow{\mathsf{load}[ret^{m'}]} b \quad b \xrightarrow{\overline{\mathsf{load}[ret^{m'}]}} a_0} \ [\text{P-Call}]$$

Figure 8 Rule for analyzing a method call.

Consider how $k$obj analyzes a method call $b = a_0.m(a_1, \ldots, a_r)$, with a target method m' resolved when a_0 points to a receiver object O. Let its $r+1$ parameters be $p_0^{m'}, \ldots, p_r^{m'}$, where $p_0^{m'}$ represents $this^{m'}$. Let its return variable $ret^{m'}$ be introduced as described in Section 4.2.2.1. Object-sensitively, $p_0^{m'}, \ldots, p_r^{m'}$ and $ret^{m'}$ are handled as if they were special

fields of O [22, 21]: $\forall\ i : a_0.p_i^{m'} = a_i$ for passing parameters and $b = a_0.ret^{m'}$ (for retrieving return values). As a result, Figure 8 gives a rule, [P-Call], for adding the PAG edges required for a method call according to [P-Load] and [P-Store]. When m' is analyzed by $k$obj, where its $this^{m'}$ variable points to O, its parameters will be initialized as $\forall\ i : p_i^{m'} = this^{m'}.p_i^{m'}$ and its return values will be made available in $this^{m'}.ret^{m'}$.

Given how $b = a_0.m(a_1, ..., a_r)$ is modeled above, we can determine whether or not a context-sensitive value flow that enters one of its invoked methods via a parameter can also exit it via another parameter without actually analyzing the invoked method itself, by enforcing $L_3(a_i, a_j)$ conservatively, i.e., ensuring that whatever flows into a_i flows also into a_j, if necessary. As will be clear in Section 4.2.2.3 below, $b = a_0.m(a_1, ..., a_r)$ needs to be approximated this way if a_0 may point to at least one cs-likely object and can be ignored otherwise.

▶ **Lemma 8.** *Let G be the PAG built by the rules in Figures 5, 7 and 8 for a method M (where how its parameters are modeled is irrelevant here). When analyzing a call $b = a_0.m(a_1, ..., a_r)$ contained in M, $L_3(a_i, a_j)$ is established iff a_0 points to at least one cs-likely object.*

Proof. Let O be an object pointed by a_0. By [P-Call], passing a_i and a_j to a target method m' at the call site is modeled by two stores $a_0.p_i^{m'} = a_i$ and $a_0.p_i^{m'} = a_j$. Thus, we have:

$$flows \Longrightarrow^{+} \ldots\ a_i \xrightarrow{\text{store}} a_0\ \overline{flows}\ O \cdots O\ flows\ a_0 \xrightarrow{\overline{\text{store}}} a_j\ \ldots \tag{14}$$

As a result, $L_3(a_i, a_j)$ is established (as far as this particular call site is concerned, regardless of its truthhood established elsewhere) iff O is a cs-likely object, in which case the "$\cdots$" that sits between the two occurrences of O can be replaced by $\xrightarrow{\text{cs-likely}}$. ◀

4.2.2.3 Approximating the Incoming Value Flows from Parameters

We discuss now how to handle the parameters of a method when it is analyzed. It is not computationally feasible to formulate our pre-analysis for a method in terms of L_3 directly (even after its parameters are modeled in a certain way). As L_3 is a CFL (with balanced parentheses), the worst-time complexity for finding the points-to set of a variable is $O(N^3\Gamma_{L_3}^3)$, where N is the number of nodes in the PAG and Γ_{L_3} is the size of L_3 [26, 15].

We now over-approximate L_3 by turning it into a regular language L_4 defined by:

$$L_4 : \begin{cases} flows \longrightarrow (\text{new} \mid \text{assign} \mid \text{load})^*((\text{store} \mid \overline{\text{store}})\ \overline{flows})? \\ \overline{flows} \longrightarrow (\overline{\text{new}} \mid \overline{\text{assign}} \mid \overline{\text{load}})^*(\text{cs-likely}\ flows)? \end{cases} \tag{15}$$

▶ **Lemma 9.** *Let G be the PAG built by the rules in Figures 5, 7 and 8. $L_4 \supseteq L_3$.*

Proof. L_4 is regularized from L_3 by no longer distinguishing store and $\overline{\text{store}}$. ◀

Thus, we are now even more conservative in abstracting aliases in L_4 than in L_3. If we replace `p.f = u` with `u.f = p` in Figure 6, $L_3(\texttt{O}, \texttt{v})$ will not hold but $L_4(\texttt{O}, \texttt{v})$ will, since

$$\texttt{O} \xrightarrow{\text{new}} \texttt{u} \xrightarrow{\overline{\text{store}}} \texttt{p} \xrightarrow{\overline{\text{new}}} \texttt{A} \xrightarrow{\text{cs-likely}} \texttt{A} \xrightarrow{\text{new}} \texttt{p} \xrightarrow{\text{assign}} \texttt{q} \xrightarrow{\text{load}} \texttt{v} \tag{16}$$

We are now ready to describe our final regular language L_5 used to decide if a variable/object in a method should be context-sensitive or not. By exploiting the fact that store and

$$\frac{p \text{ is a parameter}}{p \xrightarrow{\mathsf{param}} p \quad p \xrightarrow{\overline{\mathsf{param}}} p} \text{ [P-Param]}$$

Figure 9 Rule for adding the PAG edges for parameters.

$\overline{\mathsf{store}}$ are treated identically in L_4, we have obtained L_5, requiring the two self-loop edges to be added for each parameter of a method according to a rule, [P-Param], given in Figure 9:

$$L_5 : \begin{cases} s \longrightarrow \mathsf{param}\ \mathit{flows} \\ \mathit{flows} \longrightarrow (\mathsf{new} \mid \mathsf{assign} \mid \mathsf{load})^*((\mathsf{store} \mid \overline{\mathsf{store}})\ \overline{\mathit{flows}})? \\ \overline{\mathit{flows}} \longrightarrow (\overline{\mathsf{new}} \mid \overline{\mathsf{assign}} \mid \overline{\mathsf{load}})^*(\mathsf{cs\text{-}likely}\ \mathit{flows})? \\ \overline{\mathit{flows}} \longrightarrow \overline{\mathsf{param}}\ e \\ e \longrightarrow \epsilon \end{cases} \tag{17}$$

We can now analyze a method without knowing what its parameters may point to, by treating it effectively as a parameterless method, so that all the results developed so far are applicable.

▶ **Lemma 10.** *Let G be the PAG built for a method by the rules in Figures 5 and 7–9. Let P_1 and P_2 be its two (not necessarily different) parameters. Then $L_4(P_1, P_2) \iff L_5(P_1, P_2)$.*

Proof. Follows straightforwardly by noting the minor differences in their productions. ◀

As discussed in Section 4.2.1, if L is a CFL, $L(u, v)^n$ holds if $L(u, v)$ holds due to an L-path that contains a node n. Thus, CI_{Turner} that appears in (2) can now be defined as:

$$CI_{\text{Turner}} = \{n \mid M \in \mathbb{M}, n \text{ is a node in } G_M, \nexists P_1, P_2 \in \mathit{param}(M) : L_5^{G_M}(P_1, P_2)^n\} \tag{18}$$

where $\mathit{param}(M)$ is the set of parameters of a method M and L_5 is superscripted with the PAG, G_M, built for M. By construction, $\mathsf{CI}^{\text{OBS}}_{\text{Turner}} \subseteq CI_{\text{Turner}}$ holds due to the absence of a self-loop edge, labeled cs-likely, around each object in $\mathsf{CI}^{\text{OBS}}_{\text{Turner}}$. In addition, $\mathbb{G} \subseteq CI_{\text{Turner}}$. However, all the global variables will be context-insensitive according to (1) regardless.

Let us apply Turner to the four examples in Figure 1 to see how it has successfully selected `x` to be context-sensitive (where "return `x`" in each example has been replaced by "`this.ret = x`" and the object `A` created in Figure 1(d) is assumed to be a cs-likely object):

- **Figures 1(a) and 1(b):** $L_5(\mathtt{p}, \mathtt{this})^{\mathtt{x}}$: $\mathtt{p} \xrightarrow{\mathsf{assign}} \mathtt{x} \xrightarrow{\mathsf{store}} \mathtt{this}$.
- **Figure 1(c).** $L_5(\mathtt{p}, \mathtt{q})^{\mathtt{x}}$: $\mathtt{p} \xrightarrow{\mathsf{assign}} \mathtt{x} \xrightarrow{\overline{\mathsf{store}}} \mathtt{q}$.
- **Figure 1(d):** $L_5(\mathtt{this}, \mathtt{this})^{\mathtt{x}}$: $\mathtt{this} \xrightarrow{\overline{\mathsf{store}}} \mathtt{x} \xrightarrow{\overline{\mathsf{new}}} \mathtt{A} \xrightarrow{\mathsf{cs\text{-}likely}} \mathtt{A} \xrightarrow{\mathsf{new}} \mathtt{x} \xrightarrow{\mathsf{store}} \mathtt{this}$.

Finally, we show that Turner is precision-preserving if Observation 3 is precision-preserving. The basic idea is to show that if a variable/object is context-sensitive according to Lemma 5, i.e., Fact 2 (Figure 1), then it must reside on an L_5-path.

▶ **Theorem 11.** *Suppose Observation 3 is precision-preserving. Let G be the PAG built for a method M (Figures 5 and 7–9). If a variable/object n in M is context-sensitive by Lemma 5, then $L_5(P_1, P_2)^n$, where P_1 and P_2 are two (not necessarily different) parameters of M.*

Proof. Our proof proceeds in the following three steps:

1. We assume that M is analyzed equivalently under one cs-likely receiver object, O_M. Let M' be obtained from M by augmenting it with (1) "this^M = **new** T // O_M" and (2) "$P = \mathit{this}^M.P$" for every parameter P of M. Let G' be the resulting PAG augmented from

G. For every parameter P of M, we now have $P \xrightarrow{\overline{\text{assign}}} this^M \xrightarrow{\overline{\text{new}}} O_M \xrightarrow{\text{cs-likely}} O_M \xrightarrow{\text{new}} this^M \xrightarrow{\text{assign}} P$. Thus, $L_5(P_1, P_2)^n$ holds over G, where P_1 and P_2 are two parameters of M iff $L_5(P', P')^n$ holds over G', where P' is a parameter of M. In L_5, every variable will now be guaranteed to point to at least one object, which can be O_M.

2. We show now that all the context-sensitive value flows that enter M under its different calling contexts are tracked in L_5 if they pass through a method call $b = a_0.m_0(a_1, ..., a_r)$ (via $a_0, ..., a_r$). Thus, it suffices to consider each call site in M in isolation. Note that the loads and stores in a program can always be modeled as getters and setters.
   By Lemmas 9 and 10, Lemma 8 applies also to L_5: $L_5(a_i, a_j)$ is established in analyzing $b = a_0.m_0(a_1, ..., a_r)$ iff a_0 points to at least one cs-likely object. Thus, we only need to argue that if a_0 points to only context-insensitive objects, recorded in F_{a_0}, then each invoked method at this call site can be ignored in this sense. In this case (where $O_M \notin F_{a_0}$ as O_M is context-sensitive by construction), if some pointed-to objects of a_0 are missing in F_{a_0} (as our pre-analysis is intra-procedural), then there must exist a call chain, $a_0 = x_1.m_1(...), x_1 = x_2.m_2(...), ..., x_{t-1} = x_t.m_t(...)$ (modeled effectively as $a_0 = x_1 = ... = x_t$ in L_5), where all the pointed-to objects of x_t in the program are found intra-procedurally (under the assumption that all the receiver objects of M are abstracted by one single context-sensitive object, O_M, as explained in Step 1).
   Since Observation 3 is assumed to be precision-preserving, the value flows that enter M under its different calling contexts (i.e., receiver objects) need not be tracked, i.e., separated context-sensitively at each call site $m_i()$. To prove this claim inductively, let us write $x_{-1} = x_0.m_0(...)$ to represent $b = a_0.m_0(...)$. Now, let R_{m_i} be the set of objects returned by $m_i()$ but missed by L_5, as $m_i()$ is not analyzed. Our claim is true for $x_{t-1} = x_t.m_t(...)$, since all the objects pointed to by x_t in the program are context-insensitive. This also implies that the objects in R_{m_t} are all conflated under different calling contexts of M. Suppose that our claim holds for $m_i()$, in which case, the objects in R_{m_i} are all conflated. Let us consider $x_{i-2} = x_{i-1}.m_{i-1}(...)$. As x_{i-1} can only point to either some context-insensitive objects in F_{a_0} found intra-procedurally by L_5 or the conflated objects in R_{m_i}, our claim must also be true for $m_{i-1}()$.
3. If a variable n is context-sensitive by Lemma 5, there must exist a cs-likely O due to Step 1 such that $L_1(O, P)^n : O \textit{ flows } n' \xrightarrow{\text{store}} P$, which contains n, where n' is a variable (which may be n) and P is a parameter of M. By applying Lemmas 6 – 10 and the result established in Step 2, we must have $L_5(O, P)^n : O \textit{ flows } n' \xrightarrow{\text{store}} P$ (passing through n). As a result, $L_5(P, P)^n : P \xrightarrow{\overline{\text{store}}} n' \overline{\textit{flows}}\ O \xrightarrow{\text{cs-likely}} O \textit{ flows } n' \xrightarrow{\text{store}} P$ holds. If an object n is context-sensitive by Lemma 5, $L_5(P, P)^n$ can be established similarly. ◀

4.2.2.4 Computing CI_{Turner} with a DFA

We give an efficient algorithm for computing CI_{TURNER} with a DFA (Figure 10) obtained equivalently from the regular grammar for L_5. Our algorithm proceeds in linear time of the number of nodes in the PAG by exploiting an antisymmetric property in our DFA.

The DFA is a quintuple $\mathcal{A} = (Q, \Sigma, \delta, s, e)$, where $Q = \{s, \textit{flows}, \overline{\textit{flows}}, e\}$ is the set of states, $\Sigma = \{\text{param}, \overline{\text{param}}, \text{new}, \overline{\text{new}}, \text{assign}, \overline{\text{assign}}, \text{load}, \overline{\text{load}}, \text{store}, \overline{\text{store}}, \text{cs-likely}\}$ is the alphabet, $\delta : Q \times \Sigma \mapsto Q$ is the state transition function, s is the start state, and e is the accepting, i.e., final state.

▶ **Definition 12.** *Given a PAG edge $n_1 \xrightarrow{\sigma} n_2$ with a corresponding state transition $\delta(q_1, \sigma) = q_2$, we define $(n_1, q_1) \rightarrowtail (n_2, q_2)$ as a one-step transition. The transitive closure of $\rightarrowtail$, denoted by $\rightarrowtail^+$, represents a multiple-step transition.*

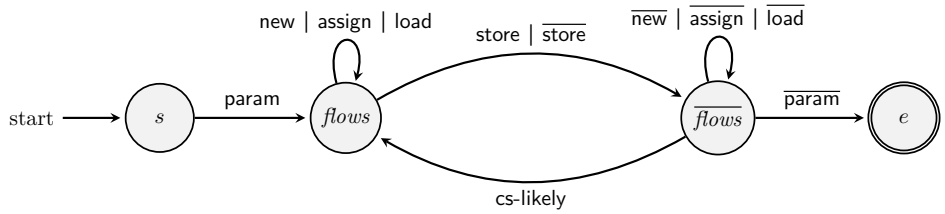


Figure 10 The DFA as an equivalent representation of the grammar for defining L_5.

We describe an antisymmetric property of our DFA in Lemmas 13 and 14 below.

▶ **Lemma 13.** *Let n_1 and n_2 be two PAG nodes. We have (1) $(n_1, s) \rightarrowtail^+ (n_2, flows) \implies (n_2, \overline{flows}) \rightarrowtail^+ (n_1, e)$ and (2) $(n_1, s) \rightarrowtail^+ (n_2, \overline{flows}) \implies (n_2, flows) \rightarrowtail^+ (n_1, e)$.*

Proof. To prove (1), we note that n_1 *flows* $n_2 \implies n_2$ $\overline{flows}$ n_1 in L_5. To prove (2), we note that n_1 *flows* $n \xrightarrow{\text{store} \mid \overline{\text{store}}} n_2 \implies n_2 \xrightarrow{\overline{\text{store}} \mid \text{store}} n$ $\overline{flows}$ n_1 in L_5, where n is a PAG node. ◀

▶ **Lemma 14.** *Let n_1 and n_2 be two PAG nodes. We have $(n_2, \overline{flows}) \rightarrowtail^+ (n_1, e) \implies (n_1, s) \rightarrowtail^+ (n_2, flows)$ and $(n_2, flows) \rightarrowtail^+ (n_1, e) \implies (n_1, s) \rightarrowtail^+ (n_2, \overline{flows})$.*

Proof. Proceeds as in the proof of Lemma 13 by noting [P-Param] given in Figure 9. ◀

In (18), we include a variable/object n in a method M (with its PAG denoted by G_M) into CI_{Turner} if $L_5^{G_M}(P_1, P_2)^n$ does not hold for any two parameters P_1 and P_2 of M. In terms of our DFA, $L_5^{G_M}(P_1, P_2)^n$ holds iff $(P_1, s) \rightarrowtail^+ (n, q) \rightarrowtail^+ (P_2, e)$, where $q \in \{flows, \overline{flows}\}$.

The antisymmetric property of our DFA is exploited below.

▶ **Theorem 15.** *Let n be a variable/object in a method with P_1 and P_2 as its two parameters. $(P_1, s) \rightarrowtail^+ (n, q) \rightarrowtail^+ (P_2, e) \iff (P_2, s) \rightarrowtail^+ (n, \overline{q}) \rightarrowtail^+ (P_1, e)$, where $q \in \{flows, \overline{flows}\}$.*

Proof. Lemmas 13 and 14. ◀

As a result, we have designed an efficient algorithm for verifying $L_5^{G_M}(P_1, P_2)^n$ by verifying $n \in R_M(flows) \cap R_M(\overline{flows})$ for a method M (with G_M as its PAG), in which, $R : Q \mapsto \wp(\mathbb{V} \cup \mathbb{H})$ returns a set of nodes in G_M reached at a given state $q \in Q$ and $R^{-1} : \mathbb{V} \cup \mathbb{H} \mapsto \wp(Q)$ is the inverse of R. These two functions are computed according to the two rules given in Figure 11. The two rules are simple: [A-I] performs the initializations needed while [A-II] computes a fixed point for each function iteratively.

$$\frac{n \in N_M}{n \in R_M(s) \quad s \in R_M^{-1}(n)} \qquad \text{[A-I]}$$

$$\frac{n_1 \xrightarrow{\sigma} n_2 \in E_M \quad q_1 \in R_M^{-1}(n_1) \quad \delta(q_1, \sigma) = q_2 \quad q_2 \notin R_M^{-1}(n_2)}{n_2 \in R_M(q_2) \quad q_2 \in R_M^{-1}(n_2)} \qquad \text{[A-II]}$$

Figure 11 Rules for computing R_M and R_M^{-1} for a method M with $G_M = (N_M, E_M)$.

Given R_M computed above, we can now obtain CI_{Turner} efficiently as follows:

$$CI_{\text{Turner}} = \{n \mid M \in \mathbb{M}, n \text{ is a node in } G_M, n \notin R_M(flows) \cap R_M(\overline{flows})\} \qquad (19)$$

4.3 Time Complexity

The worst-case time complexity of TURNER in analyzing a program is linear in terms of its number of statements, for two reasons. First, $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}}$ given in (3) and (4) can be found in $O(|\mathbb{H}|)$ based on the points-to information already computed by Andersen's analysis [1]. Second, R_M used in (19) for a method M, with its PAG denoted $G_M = (N_M, E_M)$, can be computed by the rules in Figure 11 in $O(|E_M| \times |Q|)$, where $|E_M|$ is the number of edges in G_M (constructed linearly based on the number of statements in M according to the rules in Figures 5 and 7–9) and $|Q|$, i.e., the number of states in the DFA (Figure 10), is 4.

5 Evaluation

We demonstrate that TURNER can accelerate kOBJ significantly with only negligible precision loss, by being both substantially faster than EAGLE [22] (the currently best precision-preserving pre-analysis) and substantially more precise than ZIPPER [19] (the currently best non-precision-preserving pre-analysis). We address the following three research questions:

- RQ1. Is TURNER precise?
- RQ2. Is TURNER efficient?
- RQ3. Is TURNER effective (by exploiting object containment and reachability)?

We have implemented TURNER in SOOT [42], a program analysis and optimization framework for Java, on top of its context-insensitive Andersen's pointer analysis, SPARK [17], and an object-sensitive version of SPARK (i.e., kOBJ) developed by ourselves. Our pre-analysis is implemented in under 1000 lines of Java code, which will soon be released as an open-source tool at `http://www.cse.unsw.edu.au/~corg/turner`. To compare TURNER with EAGLE [22] and ZIPPER [19], we have implemented EAGLE based on its three rules (in 600 lines of Java code) and used ZIPPER's latest version (b83b038).

As ZIPPER is evaluated in DOOP [30], we have used an experimental setting that is as close as possible to its original one in several major aspects. First, objects such as `StringBuilder`, `StringBuffer` and `Throwable` objects are merged in terms of their dynamic types and then analyzed context-insensitively as is often done in DOOP [6] and WALA [7]. Second, we perform an exception analysis together with kOBJ as in DOOP by handling exception objects caught in terms of so-called exception-catch links [5]. Third, for type-filtering purposes performed on the elements of an array, we use the declared type of its elements instead of `java.lang.Object`. Finally, we use the summaries provided in SOOT to handle native code.

We have carried out all the experiments on an Intel(R) Xeon(R) CPU E5-2637 3.5GHz machine with 512GB of RAM. We have selected a set of 12 popular Java programs, including 9 benchmarks from DaCapo2006 [3], and 3 Java applications (`checkstyle`, `JPC` and `findbugs`), which are commonly used in evaluating kOBJ [32, 40, 39, 13, 12]. The Java library used is `JRE1.6.0_45` (as the DaCapo2006 benchmarks rely only on an older version of JRE). We use TAMIFLEX [4], a dynamic reflection analysis tool, to resolve Java reflection as is often done in the pointer analysis literature [31, 32, 39, 19, 22, 21].

The time budget used for running each object-sensitive pointer analysis on a program is set as 24 hours. The analysis time of a program is an average of three runs.

Table 3 presents our main results. We compare TURNER with EAGLE and ZIPPER in terms of their efficiency and precision tradeoffs made on improving kOBJ. For each $k \in \{2, 3\}$ considered, kOBJ is the baseline, Z-kOBJ, E-kOBJ and T-kOBJ are the versions of kOBJ for performing selective context-sensitivity under ZIPPER, EAGLE and TURNER, respectively.

Table 3 Main results. For a given $k \in \{2, 3\}$, the speedups of E-kOBJ, Z-kOBJ, and T-kOBJ are normalized with kOBJ as the baseline. For all the metrics except "Speedup", smaller is better.

	Metrics	2OBJ	E-2OBJ	Z-2OBJ	T-2OBJ	3OBJ	E-3OBJ	Z-3OBJ	T-3OBJ
antlr	Time (s)	24.5	12.4	12.7	6.8	628.9	570.8	141.4	196.5
	Speedup	-	2.0x	1.9x	3.6x	-	1.1x	4.4x	3.2x
	#may-fail-casts	516	516	565	516	456	456	513	456
	#call-edges	50975	50975	51203	50975	50948	50948	51176	50948
	#poly-calls	1607	1607	1629	1607	1600	1600	1622	1600
	#avg-pts	6.110	6.110	6.585	6.125	4.927	4.927	5.427	4.945
bloat	Time (s)	412.6	290.9	324.2	138.9	10648.2	6994.7	6878.9	1902.8
	Speedup	-	1.4x	1.3x	3.0x	-	1.5x	1.5x	5.6x
	#may-fail-casts	1295	1295	1349	1295	1198	1198	1256	1198
	#call-edges	56488	56488	56988	56488	56258	56258	56837	56258
	#poly-calls	1549	1549	1587	1549	1535	1535	1577	1535
	#avg-pts	14.796	14.796	15.672	14.816	13.995	13.995	14.802	14.019
chart	Time (s)	206.2	107.5	28.3	75.1	*OoM*	12346.4	522.7	7886.1
	Speedup	-	1.9x	7.3x	2.7x	-	-	-	-
	#may-fail-casts	1339	1339	1410	1339	-	1239	1316	1239
	#call-edges	72426	72426	73009	72426	-	71987	72640	71987
	#poly-calls	1988	1988	2011	1988	-	1962	1989	1962
	#avg-pts	4.905	4.905	5.363	4.971	-	4.149	4.799	4.168
eclipse	Time (s)	10680.5	5885.3	4122.8	4686.0	*OoM*	*OoM*	*OoM*	*OoM*
	Speedup	-	1.8x	2.6x	2.3x	-	-	-	-
	#may-fail-casts	3551	3551	3718	3551	-	-	-	-
	#call-edges	162208	162208	163186	162208	-	-	-	-
	#poly-calls	9525	9525	9572	9525	-	-	-	-
	#avg-pts	17.334	17.334	19.691	17.519	-	-	-	-
fop	Time (s)	18.7	10.2	6.9	5.2	728.1	651.6	123.8	187.3
	Speedup	-	1.8x	2.7x	3.6x	-	1.1x	5.9x	3.9x
	#may-fail-casts	414	414	460	414	362	362	416	362
	#call-edges	34173	34173	34406	34173	34146	34146	34379	34146
	#poly-calls	816	816	841	816	809	809	834	809
	#avg-pts	3.577	3.577	4.132	3.597	3.359	3.359	3.942	3.383
luindex	Time (s)	15.7	9.4	6.3	4.6	596.3	532.6	131.7	185.0
	Speedup	-	1.7x	2.5x	3.4x	-	1.1x	4.5x	3.2x
	#may-fail-casts	402	402	455	402	348	348	405	348
	#call-edges	33449	33449	33689	33449	33422	33422	33662	33422
	#poly-calls	905	905	932	905	898	898	925	898
	#avg-pts	3.595	3.595	4.285	3.612	3.352	3.352	4.072	3.374
lusearch	Time (s)	22.3	15.8	11.1	10.4	1968.0	1736.8	523.5	881.1
	Speedup	-	1.4x	2.0x	2.1x	-	1.1x	3.8x	2.2x
	#may-fail-casts	417	417	473	417	366	366	425	366
	#call-edges	36247	36247	36485	36247	36220	36220	36458	36220
	#poly-calls	1103	1103	1131	1103	1096	1096	1124	1096
	#avg-pts	3.611	3.611	4.229	3.627	3.358	3.358	3.959	3.381
pmd	Time (s)	42.1	23.9	23.8	18.3	1504.0	1380.1	358.6	266.2
	Speedup	-	1.8x	1.8x	2.3x	-	1.1x	4.2x	5.7x
	#may-fail-casts	1174	1174	1252	1174	1116	1116	1199	1116
	#call-edges	59664	59664	59832	59664	59599	59599	59767	59599
	#poly-calls	2329	2329	2354	2329	2322	2322	2347	2322
	#avg-pts	4.943	4.943	6.378	4.954	4.684	4.684	5.973	4.698
xalan	Time (s)	243.2	121.8	54.2	90.9	25424.4	6771.9	694.2	1386.4
	Speedup	-	2.0x	4.5x	2.7x	-	3.8x	36.6x	18.3x
	#may-fail-casts	569	569	629	569	516	516	582	516
	#call-edges	45916	45916	46113	45916	45884	45884	46086	45884
	#poly-calls	1589	1589	1611	1589	1582	1582	1604	1582
	#avg-pts	4.253	4.253	5.258	4.272	4.096	4.096	5.014	4.119
checkstyle	Time (s)	1240.6	710.2	484.3	339.3	*OoM*	*OoM*	*OoM*	*OoM*
	Speedup	-	1.7x	2.6x	3.7x	-	-	-	-
	#may-fail-casts	1129	1129	1203	1129	-	-	-	-
	#call-edges	66702	66702	67528	66702	-	-	-	-
	#poly-calls	2188	2188	2246	2188	-	-	-	-
	#avg-pts	6.380	6.380	10.070	6.491	-	-	-	-
JPC	Time (s)	101.9	59.2	31.0	44.0	2371.1	1172.9	175.9	316.8
	Speedup	-	1.7x	3.3x	2.3x	-	2.0x	13.5x	7.5x
	#may-fail-casts	1364	1364	1438	1364	1209	1209	1281	1209
	#call-edges	81003	81003	81590	81003	79315	79315	79893	79315
	#poly-calls	4255	4255	4301	4255	4115	4115	4159	4115
	#avg-pts	5.050	5.050	5.486	5.065	4.434	4.434	4.752	4.458
findbugs	Time (s)	1820.6	681.1	128.7	150.9	*OoM*	*OoM*	2133.8	1947.0
	Speedup	-	2.7x	14.1x	12.1x	-	-	-	-
	#may-fail-casts	2037	2037	2100	2037	-	-	1884	1650
	#call-edges	87532	87532	88134	87532	-	-	87289	86599
	#poly-calls	3472	3472	3487	3472	-	-	3463	3441
	#avg-pts	8.011	8.011	8.804	8.058	-	-	7.203	6.632

Table 4 Context-sensitive facts (in millions). For all the metrics, smaller is better.

	Metrics	2obj	E-2obj	Z-2obj	T-2obj	3obj	E-3obj	Z-3obj	T-3obj
antlr	#cs-gpts	4.0K	3.8K	4.8K	2.2K	6.6K	6.0K	12.2K	2.8K
	#cs-pts	8.7M	4.9M	8.8M	1.5M	83.4M	63.4M	72.4M	33.3M
	#cs-fpts	0.4M	0.3M	0.4M	0.2M	10.2M	9.9M	10.3M	8.0M
	#cs-calls	2.4M	1.8M	1.0M	0.7M	38.5M	33.5M	6.8M	25.1M
	Total	11.5M	7.1M	10.2M	2.4M	132.1M	106.7M	89.6M	66.4M
bloat	#cs-gpts	3.2K	3.0K	4.0K	2.2K	5.1K	4.3K	11.3K	3.1K
	#cs-pts	120.4M	82.4M	111.1M	36.9M	1196.0M	856.5M	1137.5M	230.8M
	#cs-fpts	4.0M	4.0M	5.1M	3.7M	35.8M	35.4M	51.3M	30.6M
	#cs-calls	35.5M	32.1M	29.5M	15.0M	371.7M	340.5M	298.2M	109.9M
	Total	159.9M	118.4M	145.7M	55.6M	1603.6M	1232.5M	1487.0M	371.3M
chart	#cs-gpts	14.3K	13.0K	10.8K	8.2K	-	34.5K	26.3K	22.0K
	#cs-pts	64.3M	36.7M	17.0M	19.9M	-	1378.0M	171.2M	1005.7M
	#cs-fpts	1.5M	1.1M	0.8M	1.0M	-	55.4M	24.8M	48.8M
	#cs-calls	20.5M	12.2M	2.5M	8.7M	-	356.0M	23.9M	260.8M
	Total	86.4M	49.9M	20.4M	29.7M	-	1789.4M	220.0M	1315.3M
eclipse	#cs-gpts	40.6K	39.9K	28.8K	10.0K	-	-	-	-
	#cs-pts	991.9M	742.7M	744.5M	565.5M	-	-	-	-
	#cs-fpts	21.8M	21.4M	20.4M	16.2M	-	-	-	-
	#cs-calls	609.1M	342.7M	188.6M	296.5M	-	-	-	-
	Total	1622.8M	1106.8M	953.6M	878.2M	-	-	-	-
fop	#cs-gpts	3.1K	2.9K	3.7K	2.1K	4.5K	3.8K	9.8K	2.7K
	#cs-pts	3.7M	2.1M	3.6M	1.0M	70.3M	56.1M	48.8M	33.5M
	#cs-fpts	0.2M	0.2M	0.2M	0.2M	9.7M	9.4M	9.4M	7.9M
	#cs-calls	1.1M	0.9M	0.5M	0.5M	33.7M	29.8M	4.2M	25.0M
	Total	5.0M	3.2M	4.2M	1.6M	113.7M	95.3M	62.5M	66.4M
luindex	#cs-gpts	2.8K	2.6K	3.8K	1.9K	4.5K	3.9K	11.0K	2.7K
	#cs-pts	3.8M	2.2M	4.2M	1.1M	67.6M	54.2M	56.5M	33.2M
	#cs-fpts	0.2M	0.2M	0.2M	0.2M	9.7M	9.4M	10.8M	8.0M
	#cs-calls	1.1M	0.9M	0.5M	0.5M	33.1M	29.6M	4.7M	25.1M
	Total	5.2M	3.3M	4.9M	1.7M	110.4M	93.2M	72.0M	66.3M
lusearch	#cs-gpts	3.0K	2.7K	3.8K	1.9K	4.2K	3.5K	10.3K	2.5K
	#cs-pts	5.8M	3.9M	5.1M	2.2M	167.7M	151.6M	115.3M	92.2M
	#cs-fpts	0.3M	0.2M	0.2M	0.2M	11.2M	11.0M	11.0M	9.4M
	#cs-calls	2.3M	1.9M	1.0M	1.4M	108.1M	94.9M	40.5M	80.8M
	Total	8.4M	6.0M	6.4M	3.8M	287.1M	257.5M	166.9M	182.4M
pmd	#cs-gpts	3.9K	3.6K	5.9K	2.5K	5.6K	4.9K	23.8K	3.4K
	#cs-pts	12.2M	7.6M	15.1M	4.1M	144.6M	108.8M	184.5M	45.5M
	#cs-fpts	1.1M	1.0M	1.1M	0.9M	15.9M	15.3M	19.0M	11.7M
	#cs-calls	3.6M	2.6M	2.1M	1.7M	58.5M	49.0M	17.0M	33.3M
	Total	16.9M	11.1M	18.4M	6.7M	219.0M	173.1M	220.5M	90.6M
xalan	#cs-gpts	3.9K	3.6K	3.6K	2.4K	15.5K	13.5K	10.3K	6.1K
	#cs-pts	99.1M	45.9M	20.1M	14.3M	1795.3M	987.3M	253.0M	104.5M
	#cs-fpts	2.5M	2.4M	1.8M	1.9M	70.9M	63.6M	18.8M	27.0M
	#cs-calls	26.0M	19.3M	4.7M	17.2M	432.4M	300.8M	35.3M	168.1M
	Total	127.6M	67.6M	26.6M	33.3M	2298.6M	1351.7M	307.1M	299.6M
checkstyle	#cs-gpts	7.8K	7.5K	11.5K	3.9K	-	-	-	-
	#cs-pts	145.0M	107.2M	118.2M	38.0M	-	-	-	-
	#cs-fpts	2.5M	2.3M	3.0M	1.6M	-	-	-	-
	#cs-calls	78.6M	34.5M	23.2M	21.1M	-	-	-	-
	Total	226.1M	144.0M	144.4M	60.7M	-	-	-	-
JPC	#cs-gpts	7.9K	7.1K	7.7K	5.7K	22.1K	19.5K	17.5K	10.2K
	#cs-pts	28.7M	18.8M	13.9M	12.1M	618.1M	319.8M	68.6M	69.1M
	#cs-fpts	1.2M	0.9M	1.0M	0.9M	22.8M	20.0M	13.0M	13.0M
	#cs-calls	9.6M	7.1M	2.7M	5.8M	95.2M	61.4M	7.2M	38.4M
	Total	39.6M	26.9M	17.6M	18.8M	736.1M	401.3M	88.8M	120.5M
findbugs	#cs-gpts	33.5K	32.9K	10.7K	4.0K	-	-	45.6K	6.0K
	#cs-pts	326.4M	245.0M	57.2M	37.8M	-	-	545.9M	183.3M
	#cs-fpts	15.7M	15.5M	4.7M	1.1M	-	-	59.4M	26.6M
	#cs-calls	120.0M	58.3M	11.9M	9.6M	-	-	96.4M	138.5M
	Total	462.0M	318.9M	73.8M	48.5M	-	-	701.7M	348.5M

5.1 RQ1: Precision

Table 3 lists four common metrics used for measuring the precision of a context-sensitive pointer analysis [31, 41, 19, 22, 21] in terms of its context-insensitive points-to information obtained (as described in Section 2.1): (1) *#may-fail-casts*: the number of type casts that may fail, (2) *#call-edges*: the number of call graph edges discovered, (3) *#poly-calls*: the number of polymorphic calls discovered, and (4) *#avg-pts*: the average number of objects pointed by a variable, i.e., the average points-to set size.

Eagle is designed to be precision-preserving by ensuring that E-*k*obj produces exactly the same context-insensitive points-to information as *k*obj. Thus, E-2obj and E-3obj achieve trivially the same precision in all the four metrics. Zipper is designed to accelerate

$k$obj heuristically as much as possible (by also ignoring the last two value-flow patterns in Figure 1) while allowing sometimes a significant loss of precision. For 2obj, Z-2obj has caused its #avg-pts to increase by 18.1% on average, resulting in the average percentage precision losses of 7.8%, 0.7%, and 1.7% for #may-fail-casts, #call-edges, and #poly-calls, respectively. For 3obj, Z-3obj has caused its #avg-pts to increase by 16.2% on average, resulting in the average percentage precision losses of 10.8%, 0.7%, and 2.0% for #may-fail-casts, #call-edges, and #poly-calls, respectively. In this paper, Turner is designed to trade only a slight loss of precision for efficiency (by reasoning all the four value-flow patterns in Figure 1 (implicitly) using a DFA based on object containment and reachability). Despite some slightly imprecise points-to information produced (with #avg-pts increasing by 0.6% and 0.5% under T-2obj and T-3obj, respectively), both T-2obj and T-3obj preserve the precision for #may-fail-casts, #call-edges, and #poly-calls across all the 12 programs.

5.2 RQ2: Efficiency

On average, as shown in Table 3, T-$k$obj is faster than E-$k$obj but slower than Z-$k$obj. By adopting the context selections prescribed by each of the three pre-analyses, $k$obj runs faster under all the configurations. We compare Turner with Eagle and Zipper below.

- **T-kOBJ vs. E-kOBJ.** Both achieve the same precision for #may-fail-casts, #call-edges, and #poly-calls across the 12 benchmarks for $k \in \{2, 3\}$, but T-$k$obj is faster in each case. For $k = 2$, the speedups of T-2obj over 2obj range from 2.1x (for `lusearch`) to 12.1x (for `findbugs`) with an average of 3.6x. In contrast, the speedups of E-2obj over 2obj range from 1.4x (for `bloat` and `lusearch`) to 2.7x (for `findbugs`) with an average of 1.8x only. For $k = 3$, the speedups of T-3obj over 3obj range from 2.2x (for `lusearch`) to 18.3x (for `xalan`) with an average of 6.2x, while the speedups of E-3obj over 3obj range from 1.1x (for `antlr`, `fop`, `luindex`, `lusearch`, and `pmd`) to 3.8x (for `xalan`) with an average of 1.6x only. Thus, the speedups of T-$k$obj over E-$k$obj are 1.9x when $k = 2$ and 3.4x (with `chart` included even though 3obj is unscalable) when $k = 3$.
  In addition, T-$k$obj exhibits better scalability than E-$k$obj. For the four benchmarks, `chart`, `eclipse`, `checkstyle` and `findbugs`, that are unscalable under 3obj, T-3obj can now analyze `chart` and `findbugs` successfully but E-3obj can analyze `chart` only.
- **T-kOBJ vs. Z-kOBJ.** Despite its substantially better precision, T-$k$obj is faster in seven programs when $k = 2$ and three when $k = 3$. Compared with the $k$obj baseline, the average speedups achieved by T-$k$obj and Z-$k$obj are 3.6x and 3.9x, respectively, when $k = 2$, and 6.2x and 9.3x, respectively, when $k = 3$. As a result, Z-$k$obj is faster than T-$k$obj by 1.1x when $k = 2$ and 2.7x (with `chart` and `findbugs` included) when $k = 3$, on average. In terms of scalability, T-$k$obj is on par with Z-$k$obj for $k \in \{2, 3\}$.

Table 4 gives the numbers of context-sensitive facts established by $k$obj, E-$k$obj, Z-$k$obj and T-$k$obj, with #cs-gpts, #cs-pts and #cs-fpts representing the numbers of context-sensitive objects pointed by global variables (i.e., static fields), local variables and instance fields, respectively, and #cs-calls representing the number of context-sensitive call edges. In general, the speedups of a pointer analysis over a baseline come from a significant reduction in the number of context-sensitive facts computed by the baseline. For example, Z-3obj is significantly faster than T-3obj and E-3obj for `chart` as its number of context-sensitive facts is significantly less than the other two. Similarly, T-3obj is also much faster than E-3obj and Z-3obj for `bloat`. However, the analysis time of a pointer analysis is not linearly proportional to the number of context-sensitive facts computed [41]. For example, T-3obj is faster than 3obj by 3.2x for `antlr` but achieves a percentage time reduction of only 49.7%.

Table 5 Times spent by SPARK and the three pre-analyses in seconds.

	antlr	bloat	chart	eclipse	fop	luindex	lusearch	pmd	xalan	checkstyle	JPC	findbugs	Avg
SPARK	9.0	10.7	17.2	38.6	8.1	7.4	7.9	13.5	9.5	16.8	19.3	19.8	14.8
EAGLE	3.5	3.8	9.9	34.6	2.8	2.7	3.0	9.3	6.1	9.2	9.6	12.1	8.9
ZIPPER	5.4	6.5	17.1	38.9	4.4	4.2	4.6	9.5	9.0	17.9	11.5	17.4	12.2
TURNER	0.8	0.9	1.4	2.4	0.5	0.5	0.5	1.1	0.8	1.2	1.2	1.3	1.1

Table 5 gives the times spent by SPARK [17] (an implementation of context-insensitive Andersen's analysis [1]) and the three pre-analyses, EAGLE, ZIPPER and TURNER. As discussed earlier, each pre-analysis relies on the points-to information computed by SPARK to make its context selection decisions. TURNER is significantly faster than EAGLE and ZIPPER across all the 12 programs. On average, we have 1.1 seconds (TURNER), 8.9 seconds (EAGLE) and 12.2 seconds (ZIPPER). EAGLE is a single-threaded pre-analysis, ZIPPER is multi-threaded (with 16 threads used in our experiments), TURNER is currently single-threaded but is embarrassingly parallel, as it is intra-procedural. Without any parallelization, TURNER exhibits already negligible analysis times as it runs linearly in terms of the number of statements in a program.

5.3 RQ3: Effectiveness

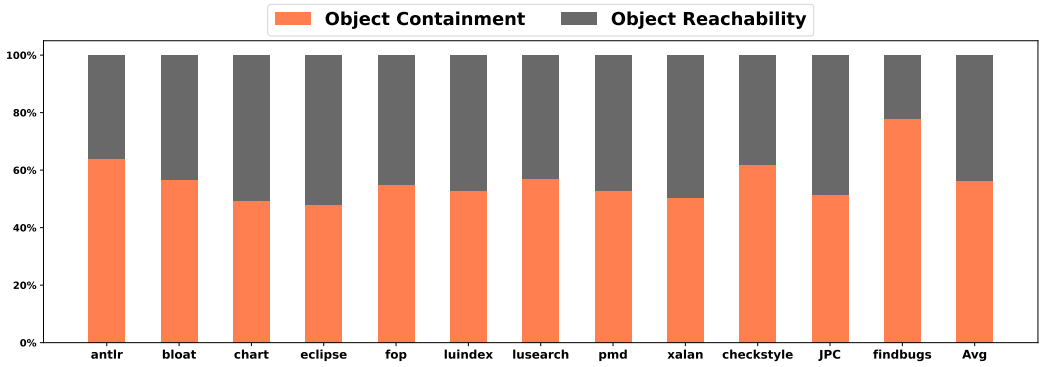


Figure 12 Percentage contributions made by TURNER's two analysis stages for the speedups of T-2OBJ over 2OBJ.

TURNER relies on object containment and reachability to make its context selections. In order to understand roughly their percentage contributions to the speedups achieved by T-kOBJ over kOBJ, let us consider two versions of T-kOBJ: (1) T-kOBJ^C, where only object containment is exploited, i.e., the objects in $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$ are context-insensitive and all the rest (the variables/objects in $(\mathbb{V} \cup \mathbb{G} \cup \mathbb{H}) \setminus \mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$) are handled as in kOBJ, and (2) T-kOBJ^R, where only object reachability is exploited by assuming $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}} = \varnothing$. Let T-$kOBJ^S_{\text{Speedup}}$ be the speedup obtained by T-kOBJ^S over kOBJ, where $S \in \{C, R, \epsilon\}$, for a program. Certainly, T-kOBJ$^C_{\text{Speedup}}$ + T-kOBJ$^R_{\text{Speedup}}$ = T-kOBJ$_{\text{Speedup}}$ is not expected for a program, as the common contribution made by T-kOBJ^C and T-kOBJ^R towards T-kOBJ$_{\text{Speedup}}$ cannot be meaningfully isolated. Instead, we consider T-kOBJ$^S_{\text{Speedup}}$/(T-kOBJ$^C_{\text{Speedup}}$ + T-kOBJ$^R_{\text{Speedup}}$), where $S \in \{C, R\}$, as the relative percentage contribution made by T-kOBJ^S towards T-kOBJ$_{\text{Speedup}}$ in order to gain a rough understanding about whether both stages are indispensable. Figure 12 illustrates the case for accelerating 2OBJ by T-2OBJ, demonstrating that both object containment and object reachability are indeed exploited beneficially for real-world programs.

Our work is largely driven by our insight stated in Observation 3. Therefore, Turner is designed to exploit both object containment and reachability to classify the objects, and consequently, the variables in a program as context-sensitive or context-insensitive.

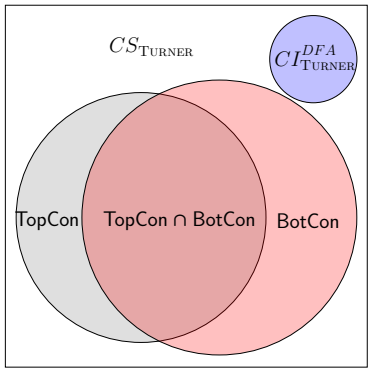


Figure 13 The Venn diagram of the objects in a program.

Figure 13 gives a Venn diagram showing how Turner classifies the containers, i.e., objects in a program. Based on object containment (Observation 3), $\mathsf{CI}^{\text{OBS}}_{\text{Turner}} = \mathsf{TopCon} \cup \mathsf{BotCon}$ gives the set of precision-uncritical, i.e., context-insensitive objects identified. Based on object reachability (performed by our DFA), $CI^{DFA}_{\text{Turner}} \subseteq \mathbb{H} \setminus \mathsf{CI}^{\text{OBS}}_{\text{Turner}}$ gives an additional set of context-insensitive sets identified. Thus, $CS_{\text{Turner}} = \mathbb{H} \setminus (\mathsf{CI}^{\text{OBS}}_{\text{Turner}} \cup CI^{DFA}_{\text{Turner}})$ represents the set of context-sensitive objects identified. On average, across the 12 programs evaluated, the ratios of $|\mathsf{CI}^{\text{OBS}}_{\text{Turner}}|$, $|CI^{DFA}_{\text{Turner}}|$ and $|CS_{\text{Turner}}|$ over $|\mathbb{H}|$ are 61.3%, 4.9%, and 33.8%, respectively. As the performance benefits of making different objects context-insensitive can be drastically different (which are hard to measure individually), these ratios, together with Figure 12, demonstrate again the effectiveness of Turner's two analysis stages.

Finally, we give two examples abstracted from the JDK library to explain why Turner does not lose any precision in #call-edges, #may-fail-casts, and #poly-calls even though it suffers from a small loss of precision in #avg-pts across the 12 programs evaluated. Turner can render some points-to sets imprecise when some top/bottom containers that are classified as precision-uncritical in $\mathsf{CI}^{\text{OBS}}_{\text{Turner}}$ should have been analyzed context-sensitively.

Figure 14 illustrates a case in which whether the object `P` created in line 4 (a top container according to Observation 3) is analyzed context-sensitively or not affects $\overline{\texttt{pts}}(\texttt{str})$ obtained in line 23. Consider 2obj, which will analyze `P` context-sensitively. When analyzing lines 19–22, we find that $\texttt{pts}(\texttt{u}i, [\,]) = \{(\texttt{U}i, [\,])\} \wedge \texttt{pts}(\texttt{U}i\texttt{.file}, [\,]) = \texttt{pts}(\texttt{P.path}, [\texttt{U}i]) = \{(\texttt{Si}, [\,])\}$, where $1 \leqslant i \leqslant 2$. When analyzing line 23, we find that $\texttt{pts}(\texttt{str}, [\,]) = \{(\texttt{S1}, [\,])\}$. Context-insensitively, 2obj thus obtains $\overline{\texttt{pts}}(\texttt{str}) = \{\texttt{S1}\}$. In the case of T-2obj, however, $\texttt{P} \in \mathsf{CI}^{\text{OBS}}_{\text{Turner}}$ will be analyzed context-insensitively instead. When analyzing lines 19–22, we have $\texttt{pts}(\texttt{u}i, [\,]) = \{(\texttt{U}i, [\,])\} \wedge \texttt{pts}(\texttt{U}i\texttt{.file}, [\,]) = \texttt{pts}(\texttt{P.path}, [\,]) = \{(\texttt{S1}, [\,]), (\texttt{S2}, [\,])\}$, where $1 \leqslant i \leqslant 2$. As `P` is context-insensitive, analyzing line 23 this time will give rise to $\texttt{pts}(\texttt{str}, [\,]) = \{(\texttt{S1}, [\,]), (\texttt{S2}, [\,])\}$. Thus, context-insensitively, T-2obj obtains $\overline{\texttt{pts}}(\texttt{str}) = \{\texttt{S1}, \texttt{S2}\}$, which contains a spurious target `S2` introduced for `str`. Despite this loss of precision in #avg-pts, however, T-2obj does not lose any precision in #may-fail-casts, #call-edges, and #poly-calls, as both `S1` and `S2` have exactly the same type, `java.lang.String`.

Figure 15 illustrates another case in which whether the object `D` created in line 14 (a bottom container according to Observation 3) is analyzed context-sensitively or not affects $\overline{\texttt{pts}}(\texttt{t})$ obtained in line 7. Consider 2obj, which will analyze `D` context-sensitively. When analyzing lines 17–20, we find that $\texttt{pts}(\texttt{v}i, [\,]) = \{(\texttt{V}i, [\,])\} \wedge \texttt{pts}(\texttt{V}i\texttt{.buffer}, [\,]) =$

```
1. class URL {
2.    String file;
3.    URL(String s) {
4.       Parts parts = new Parts(s); // P
5.       this.file = parts.getPath();
6. }
7.    String getFile() {
8.       return this.file;
9. }}
10. class Parts {
11.    String path;
12.    Parts(String p) {
13.       this.path = p;
14. }
15. String getPath() {
16.       return this.path;
17. }}

18. void main() {
19.    String s1 = new String(); // S1
20.    String s2 = new String(); // S2
21.    URL u1 = new URL(s1); // U1
22.    URL u2 = new URL(s2); // U2
23.    String str = u1.getFile();
24.    InputStream in = new FileInputStream(str);
25.    // parse content of the Stream.
26.    in.close();
27. }
```

Figure 14 Imprecise points-to information computed by T-2obj for a top container P.

```
1. class DerInputBuffer {
2.    byte[] buf;
3.    DerInputBuffer (byte[] p) {
4.       this.buf = p;
5.    }

6.    Date getTime() {
7.       byte[] t = this.buf;
8.       long l = t[0];
9.       return new Date(l);
10. }}
11. class DerValue {
12.    DerInputBuffer buffer;
13.    DerValue(byte[] buf) {
14.       this.buffer = new DerInputBuffer(buf); // D
15. }}
16. void main() {
17.    byte[] b1 = new byte[10]; // B1
18.    byte[] b2 = new byte[10]; // B2
19.    DerValue v1 = new DerValue(b1); // V1
20.    DerValue v2 = new DerValue(b2); // V2
21.    Date d1 = v1.buffer.getTime();
22. }
```

Figure 15 Imprecise points-to information computed by T-2obj for a bottom container D.

$\{(\texttt{D}, [\texttt{V}i])\} \wedge \mathsf{pts}(\texttt{D.buf}, [\texttt{V}i]) = \{(\texttt{B}i, [\,])\}$, where $1 \leqslant i \leqslant 2$. When analyzing line 7, we find that $\mathsf{pts}(\texttt{t}, [\texttt{D}, \texttt{V1}]) = \{(\texttt{B1}, [\,])\}$. Context-insensitively, 2obj thus obtains $\overline{\mathsf{pts}}(\texttt{t}) = \{\texttt{B1}\}$. In the case of T-2obj, however, $\texttt{D} \in \mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}}$ will be analyzed context-insensitively instead. When analyzing lines 17–20, we have $\mathsf{pts}(\texttt{v}i, [\,]) = \{(\texttt{V}i, [\,])\} \wedge \mathsf{pts}(\texttt{V}i\texttt{.buffer}, [\,]) = \{(\texttt{D}, [\,])\} \wedge \mathsf{pts}(\texttt{D.buf}, [\,]) = \{(\texttt{B}i, [\,])\}$, where $1 \leqslant i \leqslant 2$. As t is context-insensitive, analyzing line 7 will give rise to $\mathsf{pts}(\texttt{t}, [\,]) = \{(\texttt{B1}, [\,]), (\texttt{B2}, [\,])\}$. Thus, context-insensitively, T-2obj obtains $\overline{\mathsf{pts}}(\texttt{t}) = \{\texttt{B1}, \texttt{B2}\}$, which contains a spurious target B2 introduced for t. Despite this loss of precision in #avg-pts, T-2obj loses no precision in #may-fail-casts, #call-edges, and #poly-calls, as both B1 and B2 have exactly the same type, `java.lang.byte[]`, and in addition, each array object pointed by t is used in line 8 for obtaining a long integer only.

6 Related Work

There are two approaches for developing pre-analyses for improving the efficiency and scalability of object-sensitive pointer analysis ($k$obj) for Java: the precision-preserving approach [22, 21] and non-precision-preserving approach [32, 19, 13, 9]. Eagle [22, 21] aims to improve the efficiency of $k$obj while preserving its precision by reasoning about all the four value-flow patterns in Figure 1 implicitly via CFL reachability to make its context selections conservatively, thereby limiting the speedups achieved. In this paper, Turner addresses its limitation by trading a slight loss of precision for greater performance speedups. On the other hand, Zipper [19], as a representative non-precision-preserving pre-analysis [32, 19, 13, 9], aims to trade precision for efficiency by examining the first two value-flow

patterns in Figure 1 heuristically to make its context selections, achieving sometimes greater speedups than Eagle but at a substantial loss of precision for some programs. In this paper, Turner addresses its limitation by trading a slight loss of efficiency for greater precision by exploiting object containment (Observation 3) and then reasoning about all the four value-flow patterns in Figure 1 implicitly via an intra-procedural object reachability analysis.

There are other types of pre-analyses for $k$obj. Mahjong [39] sacrifices the precision of alias analysis (by merging objects of the same dynamic type) in order to improve the efficiency of $k$obj at a small loss of precision for a class of so-called type-dependent clients, such as call graph construction, may-fail casting, and polymorphic call detection. In contrast, Turner is designed to be a general-purpose pointer analysis to support all possible applications that rely on points-to information, including not only type-dependent clients but also alias analysis. Jeong et al. [13] apply machine learning to select the lengths of calling contexts for different methods analyzed by $k$obj for a particular client (e.g., may-fail-casting). In contrast, Turner makes its context selections by exploiting object containment and reachability.

There are also research efforts for developing pre-analyses for other programming languages. For example, Wei and Ryder [43] present an adaptive context-sensitive analysis for JavaScript. They extract user-specific function characteristics from an inexpensive pre-analysis and then apply a decision-tree-based machine learning technique to correlate these features with different types of context-sensitivity, e.g., 1-callsite, 1-object and i-th-parameter, achieving better precision and efficiency than any single context-sensitive analysis evaluated.

Elsewhere [14, 40, 12], pre-analyses are also applied to improve the precision of $k$obj at the expense of its efficiency. This thread of research is orthogonal to ours considered here.

There are other types of approaches for conducting pointer analysis in Java programs. Thiessen and Lhoták [41] propose to use context transformations rather than context strings as a new context abstraction for $k$obj, making it theoretically possible for $k$obj to run more efficiently with better precision. Instead of solving $k$obj as a whole-program analysis [17, 44, 23, 6, 18] as in this paper, demand-driven pointer analyses [37, 34, 45, 28, 38, 33] typically compute the points-to information for particular variables of interest, with call-site-sensitivity instead of object-sensitivity being often used.

Finally, Mohri and Nederhof [24] introduce an approach for over-approximating a context-free grammar (CFG) by a non-deterministic finite automaton (NFA). Prasanna *et al.* [16] adopt this approach to compute the liveness information required by a garbage collector for functional programs. For object-oriented pointer analysis, however, this is the first paper introducing an intra-procedural pre-analysis for determining selective context-sensitivity, based on a DFA over-approximated from a CFG that defines pointer analysis inter-procedurally.

7 Conclusion

We have introduced Turner, a simple, lightweight yet effective pre-analysis technique that can accelerate object-sensitive pointer analysis for Java programs with negligible precision loss. We exploit a key insight that many precision-uncritical objects in a program can be identified based on a pre-computed object containment relationship. Leveraging this approximation, we can reason about object reachability intra-procedurally to determine whether the remaining objects, together with all the variables, in the program are precision-critical or not. As a result, we have obtained a novel pre-analysis that can improve the efficiency of object-sensitive pointer analysis significantly while suffering only a small loss of precision in the points-to information produced. In particular, Turner is shown to preserve the precision of object-sensitive pointer analysis for three important clients, call graph construction, may-fail casting, and polymorphic call detection over a set of 12 popular Java programs evaluated.

We see several directions to move forward. First, we can incorporate the object allocation relationship (exploited earlier [40]) into our framework to mitigate some precision loss incurred in the scenarios shown in Figures 14 and 15. Second, we can sharpen the precision of $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}}$ with a more precise yet faster algorithm than Anderson's analysis [1]. Finally, we can analyze a method based on the context selections made earlier by exploiting a precision/efficiency tradeoff made possible by the modularity of our intra-procedural pre-analysis.

References

1 Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

2 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2666356.2594299.

3 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapobenchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1167515.1167488.

4 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250, Honolulu, HI, USA, 2011. IEEE. doi:10.1145/1985793.1985827.

5 Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, page 1–12, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1572272.1572274.

6 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 243–262, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1639949.1640108.

7 IBM T.J. Watson Research Center. WALA: T.J. Watson Libraries for Analysis, 2020. URL: http://wala.sourceforge.net/.

8 Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology*, 17(2):1–34, 2008. doi:10.1145/1348250.1348255.

9 Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, page 13–18, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3088515.3088519.

10 Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. Performance-boosting sparsification of the IFDS algorithm with applications to taint analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 267–279, San Diego, CA, USA, 2019. IEEE. doi:10.1109/ASE.2019.00034.

11 Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. Understanding and detecting evolution-induced compatibility issues in Android apps. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 167–177, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3238147.3238185.

12 Minseok Jeon, Sehun Jeong, and Hakjoo Oh. Precise and scalable points-to analysis via data-driven context tunneling. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018. doi:10.1145/3276510.

13 Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. Data-driven context-sensitivity for points-to analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):100, 2017. doi:10.1145/3133924.

14 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*, page 423–434, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2499370.2462191.

15 John Kodumal and Alex Aiken. The set constraint/cfl reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 207–218, New York, NY, USA, 2004. ACM. doi:10.1145/996893.996867.

16 Prasanna Kumar K., Amitabha Sanyal, and Amey Karkare. Liveness-based garbage collection for lazy languages. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, page 122–133, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2926697.2926698.

17 Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using spark. In *International Conference on Compiler Construction*, pages 153–169, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.5555/1765931.1765948.

18 Lian Li, Cristina Cifuentes, and Nathan Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 343–353, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2025113.2025160.

19 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Precision-guided context sensitivity for pointer analysis. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018. doi:10.1145/3276511.

20 Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. Program tailoring: Slicing by sequential criteria. In *30th European Conference on Object-Oriented Programming*, pages 15:1–15:27, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2016.15.

21 Jingbo Lu, Dongjie He, and Jingling Xue. Eagle: CFL-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity. *ACM Transactions on Software Engineering and Methodology*, 2021. To appear.

22 Jingbo Lu and Jingling Xue. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019. doi:10.1145/3360574.

23 Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005. doi:10.1145/1044834.1044835.

24 Mehryar Mohri and Mark-Jan Nederhof. Regular approximation of context-free grammars through transformation. In Jean-Claude Junqua and Gertjan van Noord, editors, *Robustness in Language and Speech Technology*, pages 153–163. Springer Netherlands, Dordrecht, 2001. doi:10.1007/978-94-015-9719-7_6.

25 Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and*

Implementation, pages 308–319, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1133255.1134018.

26 Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11-12):701–726, 1998. doi:10.1016/S0950-5849(98)00093-7.

27 Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems*, 22(1):162–186, 2000. doi:10.1145/345099.345137.

28 Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 264–274, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2259016.2259050.

29 Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences , 1978.

30 Yannis Smaragdakis. Doop-framework for Java pointer and taint analysis (using p/taint), 2021. URL: https://bitbucket.org/yanniss/doop/.

31 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*, pages 17–30, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1925844.1926390.

32 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*, pages 485–495, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2594291.2594320.

33 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for Java. In *30th European Conference on Object-Oriented Programming*, pages 22:1–22:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2016.22.

34 Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 387–400, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1133255.1134027.

35 Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J Fink, and Eran Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 196–232. Springer, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-36946-9_8.

36 Manu Sridharan, Stephen J Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–122, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1250734.1250748.

37 Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, page 59–76, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1103845.1094817.

38 Yulei Sui and Jingling Xue. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 460–473, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2950290.2950296.

39 T. Tan, Y. Li and J. Xue. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–291, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3140587.3062360.

40 Tian Tan, Yue Li, and Jingling Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium*, pages 489–510, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. doi:10.1007/978-3-662-53413-7_24.

41 Rei Thiessen and Ondřej Lhoták. Context transformations for pointer analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 263–277, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3140587.3062359.

42 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., USA, 2010. doi:10.5555/781995.782008.

43 Shiyi Wei and Barbara G Ryder. Adaptive context-sensitive analysis for JavaScript. In *29th European Conference on Object-Oriented Programming*, pages 712–734, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2015.712.

44 John Whaley and Monica S Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/996841.996859.

45 Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 155–165, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2001420.2001440.

Signal Classes: A Mechanism for Building Synchronous and Persistent Signal Networks

Tetsuo Kamina
Oita University, Japan

Tomoyuki Aotani
Mamezou Co.,Ltd., Tokyo, Japan

Hidehiko Masuhara
Tokyo Institute of Technology, Japan

Abstract

Signals are principal abstraction in reactive programming languages and constitute the basics of reactive computations in modern systems, such as the Internet of Things. Signals sometimes utilize past values, which leads to space leak, a problem where accumulated past values waste resources such as the main memory. Persistent signals, an abstraction for time-varying values with their execution histories, provide a generalized and standardized way of space leak management by leaving this management to the database system. However, the current design of persistent signals is very rudimental. For example, they cannot represent complex data structures; they can only be connected using pre-defined API methods that implicitly synchronize the persistent signal network; and they cannot be created dynamically.

In this paper, we show that these problems are derived from more fundamental one: no language mechanism is provided to group related persistent signals. To address this problem, we propose a new language mechanism *signal classes*. A signal class packages a network of related persistent signals that comprises a complex data structure. A signal class defines the scope of synchronization, making it possible to flexibly create persistent signal networks by methods not limited to the use of pre-defined API methods. Furthermore, a signal class can be instantiated, and this instance forms a unit of lifecycle management, which enables the dynamic creation of persistent signals. We formalize signal classes as a small core language where the computation is deliberately defined to interact with the underlying database system using relational algebra. Based on this formalization, we prove the language's glitch freedom. We also formulate its type soundness by introducing an additional check of program well-formedness. This mechanism is implemented as a compiler and a runtime library that is based on a time-series database. The usefulness of the language is demonstrated through the vehicle tracking simulator and viewer case study. We also conducted a performance evaluation that confirms the feasibility of this case study.

2012 ACM Subject Classification Software and its engineering → Object oriented languages; Software and its engineering → Data flow languages; Software and its engineering → Semantics

Keywords and phrases Persistent signals, Reactive programming, Time-series databases

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.17

Related Version *Extended Version*: `https://2020.splashcon.org/details/rebls-2020-papers/5/Managing-Persistent-Signals-using-Signal-Classes`

Funding This research was supported by JSPS KAKENHI Grant Number 17K00115 and 21H03418.

1 Introduction

Signals are principal abstraction in reactive programming languages [10, 17, 30, 33]. Each signal represents a data stream of a periodically updated value. By connecting them, we can declaratively specify dataflow from inputs to outputs. This mechanism was proposed as a representative construct in functional-reactive programming (FRP) [10], and has been available in imperative languages [6, 20, 30, 17, 33]. Signals constitute the basics of reactive computations in modern systems, such as the Internet of Things (IoT).

Persistent signals [18] are the abstraction for time-varying values with their execution histories *in large storage* such as a database. Unlike (transient) signals, past values of persistent signals do not disappear even after the application stops. Furthermore, persistent signals can utilize mostly "unlimited" past values. This means that persistent signals deal with the space leak problem where accumulated past values in transient signals waste resources such as the main memory. Instead, in persistent signals, space leak is managed by the underlying time-series database with large storage. This mechanism is quite useful in applications where a large amount of persistent past values is necessary, such as inspection of accidents in a vehicle tracking system and simulations based on time-series data.

Even though the idea of persistent signals was presented with their implementation and microbenchmarks, the existing persistent signals suffer from the following problems:

- Persistent signals do not provide abstractions for representing complex data structures. For example, when we implement a vehicle tracking system that records x- and y-coordinates of the running vehicle, we need to use two signals corresponding to those coordinates, instead of using one single signal representing "a vehicle." Furthermore, there are no mechanisms for ensuring the simultaneity of updates of those coordinates.
- Persistent signals can be connected only using a predefined set of API methods, which limits the use cases where persistent signals can be applied.
- Persistent signals cannot be created dynamically. This means that we cannot add any new vehicles during the execution of the vehicle tracking system. We consider this limitation to be critical.
- The structures of persistent signal networks are determined statically, and we cannot change them dynamically.

In this paper, we show that all these problems arise from a fundamental restriction: no language mechanism is provided to group and identify the related persistent signals. To address this restriction, we propose *signal classes* for the packaging mechanism of persistent signals. A set of related persistent signals (that form a persistent signal network) is grouped into one signal class. Thus, for example, we can declare "a vehicle" using a signal class where two persistent signals representing its coordinates are declared. Giving an identifier, a signal class can be instantiated dynamically, and this instance defines a scope where all persistent signals are synchronized. This instance also provides a unit of lifecycle management, which makes it easy to provide consistent management regarding lifecycle events, such as dynamic creation and destruction, as well as keeping persistency. Furthermore, by using a persistent signal whose type is also a signal class (this is only an exception to the rule that a persistent signal can have only a "primitive" type, i.e., a type that is supported by the underlying database system), we can dynamically "switch" the persistent signal networks.

We design a programming language that supports signal classes as an extension of SignalJ [17], and show how all the aforementioned problems are tackled using a simple example of vehicle tracking system. Meanwhile, we define an abstract lifecycle model of signal class instances that ensures some properties such as the bindings between persistent signals and database constructs being kept transparent to hide the underlying database from the program; e.g., multiple database tables are not created to store information regarding the same identifier.

The usefulness of the language is demonstrated through the case study of vehicle tracking simulator and viewer. In this application, a vehicle is represented as a signal class instance that encapsulates a persistent signal network comprising the dataflow from the vehicle's coordinates to its velocity. An interactive viewer is implemented using a time-oriented operation representing "scrolling back to a specific time," which is simply realized as a

declarative query on the vehicle. This application scenario implies that the proposed mechanism is useful in applications that handle a time-series data in general, in particular with an interactive user interface that shows both latest and past information. Examples include weather information, IoT sensor monitoring, and SNS timelines.

We also formalize the core language of signal classes. Because a signal class encapsulates its internal time-series data, the language needs to implicitly guarantee its internal consistency. For example, every derived signal at any timestamp must be reproduced from the values of the source signals at that timestamp. In our formalization, the computation is deliberately defined to interact with the underlying database system using relational algebra [5], and based on that, we prove the language's glitch freedom, i.e., a well-typed program does not produce any temporal inconsistencies. We also formulate its type soundness by introducing an additional check of program well-formedness.

We implemented a compiler of signal classes where a signal class is translated into a normal SignalJ class that accesses the runtime library of persistent signals. This runtime library is an extension of the existing persistent signal library [18] where we devise a mechanism that maintains the identities of signal class instances that follow the lifecycle model. This implementation is performed on a time-series database. A performance evaluation is conducted based on this implementation, and its result indicates that the vehicle tracking example is realistic in this implementation.

Contributions of this paper is listed as follows:

- Signal classes that tackle all the aforementioned problems of persistent signals, as well as their instances' lifecycle model, are developed.
- The usefulness of signal classes is demonstrated through the vehicle tracking simulator and viewer case study.
- A core language of signal classes is formalized based on relational algebra (describing the integration between signals and a time-series database) with the proofs of its glitch freedom and type soundness.
- Signal classes are implemented on the basis of a time-series database, which is proven to be responsive through a performance evaluation.

Structure of this paper. Section 2 provides some technical premises on which the proposal of signal classes is based. Section 3 discusses the difficulties in realizing persistent signals. Section 4 introduces signal classes with the descriptions of the aforementioned lifecycle model. The mechanism is explained using the vehicle tracking case study. Section 5 provides the formal definition of the core language with the proofs of glitch-freedom and type soundness. Section 6 illustrates how signal classes are implemented, and shows its performance evaluation results. Section 7 discusses related work, and Section 8 finally concludes this paper.

2 Technical Premises

2.1 Signals

Signals are abstractions for time-varying values that can be declaratively connected to form dataflows. Signals directly represent dataflows from inputs given by the environment to outputs that respond to the changes in the environment. This feature is useful in inplementing modern reactive systems, such as IoT applications. For example, assuming that the power difference of an actuator is calculated by the function `f` that takes a sensor value as an input, both the power difference and the sensor value can be represented as

signals: `powerDifference` and `sensorValue`, respectively. We describe these signals using SignalJ [17], an extension of Java that supports signals.

```
signal int sensorValue = 2000; //initial value
signal int powerDifference = f(sensorValue);
```

These declarations specify that the value of `powerDifference` is recalculated every time the value of `sensorValue` is updated.

Although signals were first proposed in several functional languages such as Fran [10], FrTime [6] and Flapjax [20] introduced their ability to imperatively change signals, and SignalJ also supports this feature using an assignment expression. For example, we can imperatively update the value of `sensorValue`, which is automatically propagated to `powerDifference`.

```
sensorValue = readFromSensors(); //powerDifference is also updated.
```

We note that the dependency between `powerDifference` and `sensorValue` is fixed during the execution, i.e., reassignment of a value to `powerDifference` is not allowed. In SignalJ, imperative update is allowed only for signals that do not depend on other signals, such as `sensorValue`.

SignalJ supports signals with a class type, and does not perceive each internal state change as a distinct update of that signal. Instead, a signal update is perceived only when the identity of the object changes. For example, assuming that the following class `C` declares a signal, namely, `s`, as its field, an assignment to `s` is not considered as a change in the signal of type `C`. We can update that signal by assigning a new instance to that:

```
signal C c = new C("A"); // Class C declares a set of signals.
c.s = 44; // assignment to the internal signal is not propagated to "c"
c = new C("B"); // reassignment to "c" results in the "switched" network
```

This enables SignalJ to perform the switching of signal networks by encapsulating a network of signals as a class instance.

Besides this mechanism to specify the dependency between time-varying values, SignalJ provides specific features that are intensively used throughout this paper. First, in SignalJ, a signal is used anywhere a non-signal is expected. Thus, in the above example, the function `f` can be a method that does not accept a signal but just an integer value. Thus, we can connect signals using legacy library methods that do not support signals, and the dependency between `sensorValue` and `powerDifference` is determined statically. Secondly, in SignalJ, a signal implicitly implements some API methods. One example of such an API method is `subscribe`, which registers an event handler that is called when the receiver signal is updated. In the following section, we will see that query API methods for persistent signals are also provided in this way.

2.2 Persistent signals

One important building block for modern reactive systems is to store time-series data, which are the histories of time-varying values comprising the reactive system. We explain this using the example of a vehicle tracking system [18]. This system records the position of each vehicle, which is obtained from automotive devices. The position changes while the vehicle is moving. In other words, the position of the vehicle is a time-varying value. There are also some other time-varying values that depend on the position, such as the estimated velocity and the total traveled distance of that vehicle. These dependencies on time-varying values motivate us to develop the system using signals. This vehicle tracking system also allows for post analysis (e.g., inspecting the cause of a car accident) and simulation. This means that the change history of each time-varying value stored in the time-series database is necessary.

Table 1 Persistent signal API (selective). In this table, we use T as a type parameter. For example, signal[T] is a type T whose modifier includes `signal`. (Adapted from [18]).

Type	Signature	Description
Basic	`within( java.sql.Timestamp ts, String interval)`	Time-series data within the extent specified by a time-stamp `ts` and `interval` representing its interval
	`bucket(String interval)`	Time-series data using the sampling rate specified by `interval`
Analytic	`avg()`, `max()`, `min()`, etc.	Average, maximum, and minimum value of the receiver signal, resp.
Domain specific	`lastDiff(int i)`	Difference between the current value of the receiver signal and the `i`th value since the last value of that signal
	`distance(signal[T] s)`	Distance between the receiver signal and `s`

Persistent signals [18] are abstractions for time-varying values with their execution histories. A persistent signal is declared as a variant of signals that encapsulates details of its execution history, which is stored in the underlying database. Queries on this execution history are supported by API methods equipped with persistent signals in advance. Each call of the API method is internally translated to the corresponding database query. Because the management of the history is left to the database system, persistent signals solve the space leak problem, where accumulated histories waste resources such as the main memory. Furthermore, persistent signals make their histories available even after the application stops.

In SignalJ, a signal is declared as a persistent signal using the modifier `persistent`. In the following example, `car1234_x` and `car1234_y` are declared as persistent signals whose time-varying values are of type `int`.

```
persistent signal int car1234_x, car1234_y;
signal int c12x = car1234_x.within(Timeseries.now, "12 hours");
signal int c12y = car1234_y.within(Timeseries.now, "12 hours");
```

In this example, these persistent signals represent the position of a specific vehicle; `car1234_x` represents the x-coordinate and `car1234_y` represents the y-coordinate.

Persistent signals are equipped with several query API methods, which are summarized in Table 1. For example, the `within` method shown above returns another persistent signal that contains all the receiver's values that have been recorded within the specified period (the past 12 hours in the above example). In other words, the return value of `within` (`c12x` or `c12y` in the above example) is a *view* of the receiver of `within`. We call such a persistent signal a *view signal*.

View signals are also used to avoid glitches among signals related with the transitive dependency (instead, temporal consistency between signals like `car1234_x` and `car1234_y` must manually be handled by the programmer). SignalJ supports *pull-based* signals, which means that a signal is re-evaluated whenever it is accessed (and it is guaranteed to be glitch-free). This strategy is also applied to the construction of view signals. Each view signal refers to a view that is created by a `SELECT` SQL query corresponding to an API method in Table 1. This query is executed on-demand when the view signal is accessed.

One particular feature of the current implementation of persistent signals is its timing of table and view generation; they are generated at compile time. In other words, lifecycle of persistent and view signals is not considered in the prior work.

2.3 Time-series databases

Time-series databases are used to implement persistent signals. They are usually specialized to store time-series data, and they provide a compact representation and convenient time-oriented API for time-series data. Because time-series data are very common in modern applications, there have been intensive research efforts in this area [16, 8, 26, 1, 19]. There are also many industrial and/or open source implementations of time-series databases, such as TimescaleDB[1], OpenTSDB[2], and InfluxDB[3].

The aforementioned implementation of persistent signals uses TimescaleDB, which is an extension of PostgreSQL, as a backend. Persistent signals have several specific properties: each record has a timestamp; once inserted, entries are not normally updated; and recent entries are more likely to be queried. To effectively interact with such time-series data, TimescaleDB provides an abstraction of a single continuous table across all space and time intervals; this is called a hypertable. All interactions with TimescaleDB (such as SQL queries) are implicitly with hypertables. The preliminary experiments on persistent signals indicate that the existing implementation is sufficiently responsive in most cases.

3 Challenges

The current design of persistent signals suffers from several problems. First, persistent signals do not provide abstractions for representing complex data structures. This problem is indeed illustrated by the aforementioned vehicle tracking example, where the position of the vehicle is represented by two distinct persistent signals, namely, `car1234_x` and `car1234_y`. This is because the persistent signals are only supported with primitive types. Thus, we cannot represent "a vehicle" as one single persistent signal, and the correspondence between x- and y-coordinates and even their synchronization must be manually written by the programmers. This imposes programming with row-level abstractions on the programmers, which is error prone.

Secondly, a view signal must be defined using an API method prepared in advance, where its SQL correspondence is defined. This is because it is difficult to derive a SQL query that creates a view from an arbitrary Java expression. However, this restriction limits the use cases where the persistent signals can be applied. For example, in the vehicle tracking system, we may want to calculate the distance to the destination as follows (assuming that `Position` is a legacy class that is not a part of the persistent signal library):

```
Position target = new Position(..);
signal double dist = target.getDistance(car1234_x,car1234_y);
```

The variable `dist` represents a time-varying value, as its depends on signals `car1234_x` and `car1234_y` (as mentioned above, even though `getDistance` does not expect signals as its arguments, SignalJ can construct a signal network that connects `dist` with `car1234_x` and `car1234_y`). It is also useful if we can use the update history of `dist` derived from database tables for `car1234_x` and `car1234_y`. This is unfortunately difficult because deriving the view is not defined in `getDistance`.

A more serious problem is that persistent signals cannot be created dynamically. This is because the database schema corresponding to persistent signals is determined by the compiler. This approach makes it easy to implement the bindings between persistent signals

[1] `https://www.timescale.com`
[2] `http://opentsdb.net`
[3] `https://www.influxdata.com`

and database constructs because database constructs already exist before the application is running, and their identities do not change during the execution. However, this is a relatively strict limitation. In the vehicle tracking example, this means that every vehicle to be tracked must be statically identified, and we cannot add any vehicles after the application is running. We consider this limitation unacceptable for real applications.

Furthermore, structures of persistent signal networks cannot be changed dynamically. This is because we cannot use a network of persistent signals as a first class citizen. In SignalJ, a signal is always evaluated to the current value (instead of obtaining "the signal itself") when it is accessed. With this semantics, SignalJ supports the switching of signal networks by encapsulating a network of signals as a class instance. However, persistent signals cannot perform this switching because they cannot have a class type.

To understand these problems, we elaborate on the details of them. First, to represent a persistent signal of "a vehicle," we might consider a complex-type persistent signal. For example, Kamina and Aotani noted that existing object-relation mapping might be applied to implement persistent signals with complex types [18], which seems to be straightforward. We define an object (e.g., a vehicle) as a persistent signal, where its internal state changes are considered its execution history maintained by the database table. This might be achieved by defining the mapping from the "`Vehicle`" class to the relation.

Unfortunately, this approach is not as easy as expected. One problem is that there might be deep nesting of internal states where one property of the internal state is another object with a complex type. Furthermore, this contradicts the SignalJ semantics where a signal update is perceived only when the identity of the object changes.

Second, the reason why we want `dist` to be implemented using a view is that we want to update its history simultaneously with updates of `car1234_x` and `car1234_y`. We might consider another approach where every update of `dist` is stored in a separated database table, i.e., `dist` is not a view signal but another persistent signal whose update history is recorded every time `car1234_x` and `car1234_y` are updated. This means that the value of `dist` is no longer calculated by the database query statement creating the view. Instead, the update of `dist` is *pushed*, which is synchronized with pushes of `car1234_x` and `car1234_y`. Thus, we must keep track of which persistent signals are synchronized. For example, there may be a number of vehicles whose updates are independent from others, and we must share the timestamp at the update only among the related persistent signals. The prior work [18] proposes the syntax for parallel assignment of persistent signals to specify the group of synchronous updates; however, intensive use of such a specific syntax makes the program very clumsy.

Finally, the dynamic creation of persistent signals requires the lifecycle of persistent signals to be managed at runtime while supporting persistency. The identity of a dynamically created persistent signal, which is statically unknown, should be retained even after the application crashes. This is because persistent signals are used to ensure tracing of past executions. After the application restarts, this identity should be taken over by the new execution because, for example, it is desirable that the vehicles in the system can be traced again using the records, including those updated before the application stopped. It is also possible that the life of persistent signal ends when the execution history of the corresponding vehicle is no longer necessary.

To enable such lifecycle management, we must keep the consistency between the related persistent signals. For example, in the vehicle tracking system we cannot simply drop `car1234_x` as it is conceptually coupled with another persistent signal `car1234_y`. As the dependencies between persistent signals are implicit in the original work, we must analyze

```
signal class Vehicle {
  String onwer, company, name;
  Position target;

  persistent signal double x, y;
  signal double x12h = x.within(Timeseries.now, "12 hours");
  signal double y12h = y.within(Timeseries.now, "12 hours");
  signal double dx = x12h.lastDiff(1);
  signal double dy = y12h.lastDiff(1);
  signal double v = dx.distance(dy);
  signal double dist = target.getDistance(x,y);

  public Vehicle(String id, String owner, String company,
                 String name, Position target) {
    this.owner = owner; this.company = company;
    this.name = name; this.target = target;
  }
  ...
}
```

Figure 1 Declaration of vehicle using a signal class.

which persistent signal is related to another one, which might be very difficult without assuming hints from the programmers. This leads to the resignation of dynamic persistent signal management.

This observation leads to our hypothesis that all these problems are just instances of more fundamental one: no syntactical support is provided to group and identify the related persistent signals. The lack of grouping mechanism leads to the separate declarations of primitive persistent signals. Such separate declarations are the cause of implicitly constructed persistent signal networks. This implicit construction makes it difficult to identify the set of persistent signals that follow the same lifecycle and can be a unit of switching of persistent signal networks. In the following section, we show that providing this grouping mechanism actually solves all these problems.

4 Signal Classes

To provide a grouping mechanism for persistent signals, we develop a language construct "signal classes" that packages a network of persistent signals into one single class. A signal class itself represents a complex data structure using a set of (primitive) persistent signals. Furthermore, a persistent signal can also have a signal class type, which realizes the dynamic switching of persistent signal networks. By providing an identifier, a signal class can be instantiated dynamically, and the persistent signals enclosed in the signal class are also created when the instance of the signal class is created[4]. Each signal class instance also forms a unit of synchronization and lifecycle management.

[4] More precisely, when the persistent signals are created is determined by the lifecycle model explained in Section 4.2.

An example of a signal class is shown in Figure 1, which declares the "`Vehicle`" class in the vehicle tracking system. A signal class is declared using the modifier `signal` in the class declaration. A persistent signal is declared using the modifier `persistent`, as in the prior work, but now it is declared within a signal class. In Figure 1, two persistent signals, `x` and `y`, are declared to record the position of the vehicle. We note that in this example, we assume that the position of a vehicle, which is monitored by automotive sensors, is periodically sent to a data center that records the vehicle's movement history. Each `Vehicle` instance is an agent reflecting the status of the "real" vehicle identified by the `id` parameter of the constructor. This instance is created at the data center when a new vehicle is registered to the system.

There are also six signals that depend on `x` and `y`, namely, `x12h`, `y12h`, `dx`, `dy`, `v`, and `dist`. While we can imperatively update the values of persistent signals `x` and `y`, any imperative re-assignment for signals depending on `x` and `y` are not allowed, and the values of them are calculated on-demand. Unlike the prior work, the construction of the right-hand side of those signals is not limited to the set of pre-defined API methods. For example, in the right-hand side of the signal `dist`, the receiver `target` of the method `getDistance` is not a signal on which the pre-defined API methods can be called. We assume that the right-hand side of each signal declaration is side-effect-free. For example, we can use the existing checker that checks a method with annotation `@SideEffectFree` does not produce any side-effects[5]:

```
class Position { ...
  @SideEffectFree
  public double getDistance(double x, double y) { ... }
  ... }
```

This check is also necessary to ensure the glitch-freedom (Section 5).

We can still use view signals in this setting. A view signal is a signal whose definition (i.e., the right-hand side of its declaration) is of the form $\texttt{p.m}(\overline{\texttt{e}})$, where `p` is a persistent or view signal, `m` is the name of an API method defined in advance, and each $\texttt{e}_i$ is an argument for `m`. In Figure 1, `x12h`, `y12h`, `dx`, `dy`, and `v` are view signals. We note that the value of view signal is also calculated on demand. One advantage of using view signals is it reduces the update overhead of persistent signals. Furthermore, view signals are useful to "filter" the persistent signals that contain all the execution histories managed by the underlying database system. For example, in Figure 1 we use the `within` query to filter out the old data to avoid performance degradation [18]. Two other API methods, `lastDiff` and `distance` taken from Table 1, are also used.

In summary, the behavior of the `Vehicle` instance is interpreted as follows. Once the instance, namely, `aCar`, of `Vehicle` is created, we can call the `set` method, which is an interface method that all signal classes implicitly implement, to update persistent signals `x` and `y`:

```
aCar.set(33.239148, 131.611722); // setting an initial position.
```

This `set` method first sets the value of `x` and `y` with the provided arguments and then implicitly calculates the value of `dest` using the current values of `x` and `y`. The value of each view signal is automatically determined by the database query statement that creates the corresponding view. For example, `dx` and `dy` calculate the delta between the current value and the last value for each x- and y-coordinate, respectively. The view signal `v` calculates the estimated velocity of the moving vehicle.

[5] `http://checkerframework.org`

4.1 Time-oriented queries on signal class instances

Our vehicle tracking system consists of two subsystems: the vehicle controller (simulating the vehicle's behavior by periodically calling `set`) and the vehicle viewer. Those subsystems run as two distinct processes, and each process has its own signal class instance of the same vehicle. We assume that only the controller continuously updates the vehicles and the viewer accesses each vehicle's time-series data.

To implement the viewer, we can perform time-oriented queries on signal class instances. For example, we can obtain a snapshot of the `aCar` instance at the time when it causes an accident. The following `snapshot` method provides a temporal view of `aCar` at the time specified by the timestamp provided as an argument:

```
aCar.snapshot("2018-06-01T18:10:00").v
```

This query set the internal cursor of the receiver instance to the specified timestamp, which makes the value of every persistent signal on `aCar` be calculated using values of the specified timestamp. This query is called every time the GUI slider is set to point a specific timestamp. Moreover, an argument for `snapshot` can be a variable such as a signal. For example, assuming that a variable `slider` is a signal of "currently selected timestamp using the slider," a vehicle at the time selected by the slider can simply be represented as `aCar.snapshot(slider)`.

We note that the vehicle can still continue to update its persistent signals using `set`, which can be accessed by resetting the cursor to current time using the timestamp "`NOW()`":

```
aCar.snapshot("NOW()");
```

4.2 Lifecycle of a signal class instance

We develop the lifecycle model of signal class instances. In the original work, the underlying database tables for persistent signals are generated by the compiler [18]; this forces persistent signals to be defined statically and makes it very difficult to add new persistent signals at runtime. In the proposed lifecycle model, a signal class instance can be instantiated dynamically, and thus the underlying database tables are generated at runtime. A signal class encapsulates related persistent signals into one module, and this module provides a unit of lifecycle management.

Once created, the history of a signal class instance can exist on the disk even after the application stops. Its identity is preserved on the disk, and when the application restarts, this instance becomes available again from the program. For example, consider the following declaration of a `Vehicle` instance:

```
Vehicle aCar = new Vehicle("501a1234", "Haskell", "Toyota", "Sienta");
```

If there are no database constructs on the disk that correspond to `aCar`, the `Vehicle` instance is created with fresh database constructs. If there already exist such constructs, `aCar` is simply bound to them. In this mechanism, we must keep track of this binding on the disk, and this is done using the `id` parameter, which is mandatory for every constructor in a signal class. This is used as a key to identify the signal class instance.

Figure 2 formalizes this lifecycle model using a state machine diagram. This diagram models one instance that has a full-control to its history (e.g., updating the history like the vehicle controller). There may exist other instances that only perform queries on the history (like the vehicle viewer), but the diagram simply omits such details. Each event that changes its state is triggered by environmental changes or internal program operations. Some of them can be explicitly triggered by calling the interface methods that every signal class implicitly

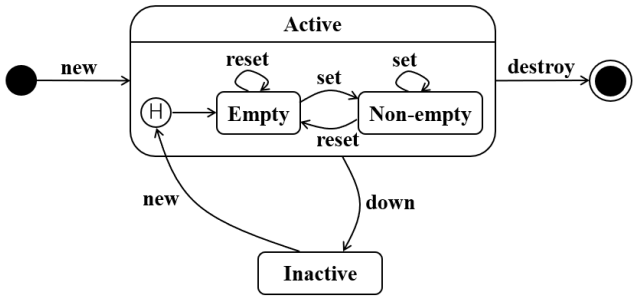


Figure 2 State machine diagram of signal class instance.

```
public interface SignalClassInstance {
  public void set(Object ... newValues);
  public void reset();
  public void destroy();
}
```

Figure 3 Interface for representing signal class lifecycle events.

implements. This interface is shown in Figure 3. We note that this listing imprecisely describes the formal parameters of `set` to indicate that the number of its formal parameters and their types are not defined in advance; the interface of `set` is implicitly derived from the persistent signals declared in that signal class. For example, `set` for `Vehicle` is declared as follows by listing the formal parameters that correspond to the persistent signals `x` and `y`:

```
  public void set(double x, double y);
```

This interface changes by definition of the signal class. The compiler translates the invocation of `set` to make it compatible with the runtime library, which provides the generalized interface.

The states in the lifecycle model are defined as follows:

Initial: The signal class instance has never been created, and there are no persistent or view signals that are bound with this signal class instance.

Active: We can access this signal class instance if it is in this state. A signal class instance will be active just after it is created using the `new` expression. This state consists of the substates **Empty** and **Non-empty**.

Empty: This is the state of a signal class instance where the histories (i.e., the data in the database) of persistent and view signals contained in it are empty. As indicated by the incoming edge from the history state "`H`", **Empty** is the initial substate of the *Active* state, which means that every signal class instance starts with empty histories. The **reset** event also makes the signal class instance empty. Some operations for signals (for example, taking a last value) cannot be performed when the signal class instance is empty.

Non-empty: Persistent and view signals contained in the signal class instance have recorded some of their execution histories. Any operations for signals can be performed when the signal class instance is non-empty.

Inactive: We cannot access this signal class instance if it is in this state. A signal class instance will be inactive if there becomes no pointers that access this instance or the application stops for some reason (e.g., maintenance or crash). A signal class instance preserves its identity on the disk even after it is removed from the main memory.

Final: The signal class instance is destroyed, and persistent and view signals in this instance no longer exist.

The events in the lifecycle model are defined as follows:

new: This event is triggered by the `new` expression, which creates a signal class instance. It generates different side-effects on the signal class instance according to its previous state. If this event occurs with the initial state, it creates new persistent and view signals in the signal class instance with their empty contents. If this event occurs with the inactive state, the signal class instance becomes active and resumes its internal substate, as indicated by the history state "`H`".

set: This event can be generated only when the signal class instance is active. It is triggered when the persistent signals contained in the signal class instance are updated. If there are multiple persistent signals (as in the case of `Vehicle`), their updates are synchronized. The signal class implicitly provides the `set` method for this synchronized update, which makes the state of the signal class instance non-empty.

reset: This event can also be generated only when the signal class instance is active. It is triggered by the `reset` method declared in Figure 3, which cleans the existing histories of the persistent and view signals. This event makes the state of the signal class instance empty.

down: This event is triggered by external or internal environmental changes; it is triggered if the signal class instance can no longer be accessed or the application stops for some reason. After this event, the signal class instance disappears. This instance can however be reactivated, like the "ship of Theseus," using the blueprint of it stored in the database, i.e., when the application restarts, the **new** event can be triggered to restore this instance.

destroy: The signal class instance completes its life when the histories of its persistent and view signals are no longer necessary, and this is performed by generating the destroy event. This event can be fired by calling the `destroy` method shown in Figure 3. It can also be generated by some external environmental changes (e.g., dropping the tables and views from the console of the database management system). After firing this event, no events can be fired on this signal class instance. We note that we can still generate the **new** event using the same identifier (passed to the `id` parameter) again, which starts another independent lifecycle of this `id`.

Importantly, every lifecycle management is performed on the basis of this model. We cannot solely generate, update, or drop the content of each persistent signal. Instead, all related signals are simultaneously generated, updated, and dropped. This makes it easy to ensure data consistency between them.

One important property of the signal class instance is that there should not be multiple signal class instances with the same `id`. This property is ensured according to the lifecycle model. This is because this model does not accept any event sequences where multiple **new** events are triggered until the next **down** or **destroy** events are issued. The **new** event must be the first event of the sequence, and it can follow only the **down** event. Thus, the signal class instance can be activated only when there are no other signal class instances with the same `id`.

4.3 Synchronized update

The vehicle controller continuously updates the histories of the running vehicles. Each signal class instance forms a unit of synchronization. Each signal class provides the `set` method for synchronized update of persistent signals. This improves the synchronized update in the original work [18], where the programmers must ensure that persistent signals that are

defined independently are updated at the same time. For example, we can define the following `run` method in the `Vehicle` class that periodically updates the position of the vehicle:

```
public void run() {
  double[] current = new double[2];
  while (true) {
    current = getGeographicCoordinatesFromSensors();
    set(current[0], current[1]);
  } }
```

This `set` method first computes the value of `dist` using `current[0]` and `current[1]`, and then inserts the triple of current timestamp, `current[0]`, and `current[1]`. Thus, all persistent and view signals are updated at once, and the programmers do not have to worry about any glitches inside the instance.

4.4 Switching network of persistent signals

In SignalJ, we can construct a signal of an object that encapsulates other signals [17]. This feature can be extended to the persistent signals: we can construct a persistent signal of a signal class instance that encapsulates other persistent signals. This allows us to construct a network of persistent signals that changes dynamically, like the "switch" in the FRP languages.

For example, we can construct a signal class that monitors a particular instance of `Vehicle`.

```
signal class Monitor {
  persistent signal Vehicle v;
  public Monitor(String id) { .. } }
```

According to the lifecycle model, we can initialize the persistent signal `v` by issuing the **set** event on the instance `m` of `Monitor`[6]

```
Monitor m = new Monitor("aMonitor");
m.set(aCar);
```

The subsequent **set** events on `m` change the instance of `Vehicle` that `m` monitors, and this change is recorded in the history that is bound with `m`. For example, we may want to monitor some suspicious vehicles more intensively, and the history of `m` is available for inspecting which vehicles were considered suspicious in the past.

We note that SignalJ's object-type signals are considered updated only when the identity of the object changes, and this property is also available in the persistent signals. This means that the persistent signal `v` in `Monitor` does not have to record the instance of `Vehicle` but only its identifier, which is provided by the programmer using the `id` parameter of the signal class constructor.

4.5 Threat to Validity

While the vehicle tracking example well describes the problems of prior work [18] and how signal classes address them, we have not performed any other empirical studies in this work. We consider that discussions in this paper also apply to other applications that handle

[6] We further discuss the initialization issue in Section 6.2.

```
CL  ::=  signal class C { PS FS M }
PS  ::=  persistent signal C p;
FS  ::=  signal C p=e;
 M  ::=  C m(C x) { return e; }
 e  ::=  x | e.p | e.set(e) | e.snapshot(t) | e.m(e) | new C(l) | l | t
 v  ::=  l | t
```

Figure 4 Abstract syntax of signal classes.

timelines of time-varying values that can be identified by some `id`s, such as SNS applications and IoT device monitoring. Further analysis on such application scenarios remain as future work.

5 Formalization

To study important properties of signal classes such as glitch-freedom, we formally define the formal semantics of signal classes based on the simplified syntax of SignalJ shown in Figure 4. The syntax is based on Featherweight Java [15]. Let the metavariables `C` and `D` range over class names; `o` and `p` range over persistent signals; `e` range over expressions; `x` range over variables, which include a special variable `this`; `l` range over identifiers; `t` range over timestamps; `v` range over values; and `m` range over method names. Overlines denote sequences, e.g., $\overline{\texttt{e}}$ represents a possibly empty sequence $\texttt{e}_i, \cdots, \texttt{e}_n$, where n denotes the length of the sequence. We write the length of sequence $\overline{\texttt{e}}$ as $\#(\overline{\texttt{e}})$. We use $\overline{\texttt{C}}\ \overline{\texttt{p}}$ as shorthand for "$\texttt{C}_1\ \texttt{p}_1 \cdots \texttt{C}_n\ \texttt{p}_n$" and $\overline{\texttt{C}}\ \overline{\texttt{s}=\texttt{e}}$ as shorthand for "$\texttt{C}_1\ \texttt{s}_1=\texttt{e}_1 \cdots \texttt{C}_n\ \texttt{s}_n=\texttt{e}_n$."

An expression can be either a variable, an access to a persistent signal, an invocation of special methods `set` and `snapshot` that correspond to the **set** event and a time-oriented query, respectively, a method invocation other than `set` and `snapshot`, an instance creation, or a value that can be either an identifier `l` or a timestamp `t`. The instance creation receives only one identifier as its argument. We assume the set *Id* of identifiers and $\texttt{l} \in \mathit{Id}$. We also assume a total order set *Time* of timestamps where $\bot \in \mathit{Time}$ and $\forall \texttt{t} \in \mathit{Time}.\ \bot \leq \texttt{t}$, i.e., $\bot$ is a bottom element.

In our proposal, there are two kinds of persistent signals: a persistent signal whose value is imperatively set by calling the `set` method, and that whose value is updated on-demand when it is accessed. Figure 4 syntactically distinguishes those two: a persistent signal declaration `PS` representing the former, and a persistent signal declaration `FS` representing the latter; its value is updated on-demand by evaluating the expression `e` that appears in the right-hand side. We note that the latter includes view signals that are calculated using API methods such as database aggregates shown in Section 2.2, as rows in a view are calculated on-demand.

We also apply another simplification to the calculus: the syntax does not provide subclassing, meaning that there are no subtyping rules in the calculus. This is a drastic simplification, but subclassing actually does not interact with the behavior of signal classes, as the execution history is stored for each instance and thus signal lookup is performed in per-instance basis, meaning that the class hierarchy is actually not used during the signal lookup.

A program $(CT, \texttt{e})$ consists of a class table CT that maps a class name `C` to a class declaration `CL` and an expression `e` that corresponds to the body of the `main` method. We

$$\frac{\texttt{signal class C \{ persistent signal } \overline{\texttt{C}}\ \overline{\texttt{p}}\texttt{; ... \}}}{\mathit{sources}(\texttt{C}) = \overline{\texttt{C}}\ \overline{\texttt{p}}}$$

$$\frac{\texttt{signal class C \{ persistent signal } \overline{\texttt{C}}\ \overline{\texttt{p}}\texttt{; ... \}}}{\mathit{signalType}(\texttt{C}, \texttt{p}_i) = \texttt{C}_i}$$

$$\frac{\texttt{signal class C \{ ... signal } \overline{\texttt{C}}\ \overline{\texttt{p}}\texttt{=}\overline{\texttt{e}}\texttt{; ... \}}}{\mathit{signalType}(\texttt{C}, \texttt{p}_i) = \texttt{C}_i}$$

$$\frac{\texttt{signal class C \{ ... signal } \overline{\texttt{C}}\ \overline{\texttt{p}}\texttt{=}\overline{\texttt{e}}\texttt{; ... \}}}{\mathit{signalExpr}(\texttt{C}, \texttt{p}_i) = \texttt{e}_i}$$

$$\frac{\texttt{signal class C \{ ... C}_0\ \texttt{m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{) \{ return e}_0\texttt{; \} \}}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}\texttt{.e}}$$

$$\frac{\texttt{signal class C \{ ... C}_0\ \texttt{m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{) \{ return e}_0\texttt{; \} \}}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \overline{\texttt{C}} \to \texttt{C}_0}$$

Figure 5 Auxiliary definitions.

assume that $CT(\texttt{C}) = \texttt{signal class C ...}$ for any $\texttt{C} \in \mathit{dom}(CT)$. We also assume that all signals and methods in the same class, and all parameters in the same method are distinct.

In the following discussion, we use the auxiliary definitions shown in Figure 5. The function *sources*(C) returns a sequence of all pairs of a signal declared with `persistent` and its type in class C. The function *signalType*(p, C) returns the type of signal p (regardless to say that it is declared with `persistent`) in class C. The functions *mbody*(p, C) returns a pair $\overline{\texttt{x}}\texttt{.e}$ of parameters and a method body of method m in class C. Similarly, *mtype*(p, C) returns a pair $\overline{\texttt{C}} \to \texttt{C}$ of parameter types and a return type of method m in class C.

5.1 Small-step semantics

We show the reduction rules of expressions in Figure 6. Those are given by the relation of the form $\mu \mid \texttt{e} \longrightarrow \mu' \mid \texttt{e}'$, which is read as "an expression e under an environment μ reduces to e′ under μ'." The environment μ is a set of mapping $\texttt{l} \mapsto \mathcal{R}_{\texttt{C(l)}}$, where l is the identifier of the signal class instance, and $\mathcal{R}_{\texttt{C(l)}}$ is an execution history of l that is a relation defined as follows:

$$\frac{\mathit{sources}(\texttt{C}) = \overline{\texttt{C}}\ \overline{\texttt{p}}}{\mathcal{R}_{\texttt{C(l)}} = (\mathit{time}, \overline{\texttt{p}})}$$

This relation is handled using the operations provided by the relational algebra [5]: $\pi_{col}(\mathcal{R})$ represents a projection of a relation $\mathcal{R}$ by *col* (i.e., selecting the column *col* from $\mathcal{R}$), and $\sigma_c(\mathcal{R})$ represents filtering $\mathcal{R}$ using the condition c. We often use a singleton set {l} and its value l interchangeably. We use the predicate *latest*, which is true only if the *time* field of the tuple has the largest value among the relation.

$$\frac{\mu(\mathtt{l}_0) = \mathcal{R}_{\mathtt{C}(\mathtt{l}_0)} \qquad \pi_{\mathtt{p}}(\sigma_{latest}(\mathcal{R}_{\mathtt{C}(\mathtt{l}_0)})) = \mathtt{l}}{\mu \mid \mathtt{l}_0.\mathtt{p} \longrightarrow \mu \mid \mathtt{l}} \quad \text{(R-PSIGNAL)}$$

$$\frac{\mu(\mathtt{l}_0) = \mathcal{R}_{\mathtt{C}(\mathtt{l}_0)} \qquad \mathit{signalExpr}(\mathtt{C}, \mathtt{p}) = \mathtt{e}}{\mu \mid \mathtt{l}_0.\mathtt{p} \longrightarrow \mu \mid \mathtt{e}} \quad \text{(R-VSIGNAL)}$$

$$\mu \mid \mathtt{new\ C(l)} \longrightarrow \mu \oplus (\mathtt{l} \mapsto \emptyset) \mid \mathtt{l} \quad \text{(R-NEW)}$$

$$\frac{\begin{array}{c}\mathcal{R}'_{\mathtt{C}(\mathtt{l})} = \{(\mathtt{t}, \overline{\mathtt{l}})\} \cup \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mathtt{t} > \sigma_{latest}(\pi_{time}(\mathcal{R}_{\mathtt{C}(\mathtt{l})})) \\ \mu(\mathtt{l}) = \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mu' = \mu \oplus (\mathtt{l} \mapsto \mathcal{R}'_{\mathtt{C}(\mathtt{l})}) \qquad \overline{\mathtt{l}} \in \mathit{dom}(\mu)\end{array}}{\mu \mid \mathtt{l.set}(\overline{\mathtt{l}}) \longrightarrow \mu' \mid \mathtt{l}} \quad \text{(R-SET)}$$

$$\frac{\begin{array}{c}\mu(\mathtt{l}_0) = \mathcal{R}_{\mathtt{C}(\mathtt{l}_0)} \qquad \mathcal{R}'_{\mathtt{C}(\mathtt{l}_0)} = \mathcal{R}_{\mathtt{C}(\mathtt{l}_0)} \setminus \sigma_{\mathtt{t} < time}(\mathcal{R}_{\mathtt{C}(\mathtt{l}_0)}) \\ \mu' = \mu \oplus (\mathtt{l}_0 \mapsto \mathcal{R}'_{\mathtt{C}(\mathtt{l}_0)}) \qquad \mathcal{R}'_{\mathtt{C}(\mathtt{l}_0)} \neq \emptyset\end{array}}{\mu \mid \mathtt{l}_0\mathtt{.snapshot(t)} \longrightarrow \mu' \mid \mathtt{l}_0} \quad \text{(R-TIME)}$$

$$\frac{\mu(\mathtt{l}) = \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mathit{mbody}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e} \qquad \mathtt{m} \neq \mathtt{set} \qquad \mathtt{m} \neq \mathtt{q}}{\mu \mid \mathtt{l.m}(\overline{\mathtt{l}}) \longrightarrow \mu \mid \mathtt{e}[\overline{\mathtt{x}}/\overline{\mathtt{l}}, \mathtt{this/l}]} \quad \text{(R-INVK)}$$

Figure 6 Small-step computation rules for expressions.

The rules R-PSIGNAL and R-VSIGNAL define how an access to a signal behaves. These rules are straightforward. An access to the persistent signal `p` results in the value in the `p` column of the latest tuple in $\mathcal{R}_{\mathtt{C}(\mathtt{l})}$. An access to a non-source signal results in the right-hand side of its declaration.

The rule R-NEW defines the reduction of the signal class instance creation; it adds the mapping from `l`, which is the identifier of the created instance, to its execution history to μ and returns `l`. We use $\oplus$ as a destructive update of the mapping, i.e., $(x \oplus y)(k) = y(k)$ if k is in the domain of y or $x(k)$ otherwise.

There are three rules for method invocation. In the rule R-SET for the call of `set`, which represents the synchronized update, we first choose a timestamp `t` that is greater than the "largest" value in *Time* and put the tuple $(\mathtt{t}, \overline{\mathtt{l}})$ into the relation. We assume that $\sigma_{latest}(\pi_{time}(\mathcal{R}))$ returns $\bot$ if $\mathcal{R} = \emptyset$. A call of `set` returns the identifier of its receiver, which allows us to describe subsequent computations on the receiver. The rule R-TIME destructively changes the relation $\mu(\mathtt{l}_0)$ so as to be filtered by the given timestamp `t` (as the calculus does not model the database cursor). We note that this is a simplification of the original language's behavior; i.e., unlike the original language, we cannot recover the time-series values that are lost by the `snapshot` operation. R-TIME requires that the resulting relation must not be empty. The rule R-INVK for method invocation (other than `set` and `snapshot`) is straightforward.

We also define the congruence rule that enables a reduction of subexpressions. For this purpose, we first introduce the evaluation context E, which is defined as follows:

$$\begin{array}{rcl} E & ::= & [] \mid E.\mathtt{p} \mid E.\mathtt{set}(\overline{\mathtt{e}}) \mid \mathtt{l.set}(\overline{\mathtt{l}}, E, \overline{\mathtt{e}}) \mid E.\mathtt{snapshot(t)} \mid E.\mathtt{m}(\overline{\mathtt{e}}) \mid \\ & & \mathtt{l.m}(\overline{\mathtt{l}}, E, \overline{\mathtt{e}}) \end{array}$$

```
signal class C1 {
  persistent signal D p;
  C2 m() { return new C2(l2).set(this); }
}
signal class C2 {
  persistent signal C1 p;
  signal C1 q = this.p.set(new D(l3).set(...));
  signal E n = this.o(this.p.p,this.q.p);
  E o(D x, D y) { ... }
}
```

Figure 7 A glitch-introducing program.

Each evaluation context is an expression with a hole (written []) somewhere inside it. We write $E[\mathsf{e}]$ for an expression obtained by replacing the hole in E with e.

Using E, the congruence rule is defined as follows:

$$\frac{\mu \mid \mathsf{e} \longrightarrow \mu' \mid \mathsf{e}'}{\mu \mid E[\mathsf{e}] \longrightarrow \mu' \mid E[\mathsf{e}']} \qquad \text{(R-Cngl)}$$

The evaluation context syntactically defines the evaluation order of subexpressions in a method invocation, e.g., the arguments are not reduced until the receiver becomes an identifier.

5.2 Static semantics

One significant research question is how the internal state of each signal class instance is kept consistent. This question is also known as an assurance of glitch-freedom. Myter et al. stated that the key intuition behind glitches is that they can only occur for certain topologies of signal networks [23], where two or more propagations from the same source signal (A) join at the other signal (B). A glitch is a situation where the value of B is calculated using A's values with different timestamps.

If no static checking is performed, a glitch can occur even in our simple calculus. Consider the signal classes `C1` and `C2` declared in Figure 7, and the main expression `new C1(l1).m().n`. The signal `n` calls the method `o` that consumes signals `C2.p` and `C2.q`. Those signals depend on the same signal `p` that is a member of `new C1(l1)`. Furthermore, the signal `C2.q` calls the `set` method when its value is accessed. This `set` updates the signal `C1.p` with a new timestamp; thus the signal `n` handles values of the same signal with different timestamps. This is a glitch[7].

Another important role of static analysis is to ensure the calculus type soundness, i.e., to avoid a situation where a program get stuck by, e.g., accessing an undefined attribute in $\mathcal{R}$. To ensure the calculus glitch-freedom and type soundness, we develop a type system of the proposed calculus.

[7] In general, a signal network can contain nodes with different timestamps. Such a network is often useful, as illustrated by the signal network constructed using `lastDiff` (Figure 1, the calculus omits this feature). A glitch is the situation where "the same node" is observed with different timestamps.

$$\frac{\Gamma = \overline{\mathtt{x}} : \overline{\mathtt{C}}}{\Gamma \mid \emptyset \vdash \mathtt{x}_i : \mathtt{C}_i}\ (\text{T-Var}) \qquad \frac{\Sigma = \overline{\mathtt{l}} : \overline{\mathtt{C}}}{\emptyset \mid \Sigma \vdash \mathtt{l}_i : \mathtt{C}_i}\ (\text{T-Id}) \qquad \frac{\mathtt{t} \in \mathit{Time}}{\emptyset \mid \emptyset \vdash \mathtt{t} : \mathtt{T}}\ (\text{T-Ts})$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{signalType}(\mathtt{C}_0, \mathtt{p}) = \mathtt{C}}{\Gamma \mid \Sigma \vdash \mathtt{e}_0\mathtt{.p} : \mathtt{C}}\ (\text{T-Signal})$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{sources}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{p}} \qquad \Gamma \mid \Sigma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}}}{\Gamma \mid \Sigma \vdash \mathtt{e}_0\mathtt{.set(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{C}_0}\ (\text{T-Set})$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \Gamma \mid \Sigma \vdash \mathtt{t} : \mathtt{T}}{\Gamma \mid \Sigma \vdash \mathtt{e}_0\mathtt{.snapshot(t)} : \mathtt{C}_0}\ (\text{T-Time})$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{C}} \to \mathtt{C} \qquad \Gamma \mid \Sigma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}}}{\Gamma \mid \Sigma \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{C}}\ (\text{T-Invk})$$

$$\Gamma \mid \Sigma, \mathtt{l} : \mathtt{C} \vdash \mathtt{new\ C(l)} : \mathtt{C}\ (\text{T-New})$$

Figure 8 Expression typing.

$$\frac{\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \mid \emptyset \vdash \mathtt{e}_0 : \mathtt{C}_0}{\mathtt{C}_0\ \mathtt{m(}\overline{\mathtt{C}}\ \overline{\mathtt{x}}\mathtt{)}\ \mathtt{\{\ return\ e_0;\ \}\ ok\ in\ C}}\ (\text{T-Method})$$

$$\frac{\mathtt{this} : \mathtt{C} \mid \emptyset \vdash \overline{\mathtt{e}} : \overline{\mathtt{D}} \qquad \mathit{sideeffectfree}(\overline{\mathtt{e}}) \qquad \overline{\mathtt{M}}\ \mathtt{ok\ in\ C}}{\mathtt{signal\ class\ C\ \{\ persistent\ signal}\ \overline{\mathtt{C}}\ \overline{\mathtt{p}}\mathtt{;\ signal}\ \overline{\mathtt{D}}\ \overline{\mathtt{o}}\mathtt{=}\overline{\mathtt{e}}\mathtt{;}\ \overline{\mathtt{M}}\ \mathtt{\}\ ok}}\ (\text{T-Class})$$

Figure 9 Method and class typing.

Typing rules for expressions are shown in Figure 8. A type environment Γ is a finite mapping from variables to class names. An identifier environment Σ is a finite mapping from identifiers to class names. A type judgment for expressions is of the form $\Gamma \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$, read as "expression `e` is given type `C` under the type environment Γ and identifier environment Σ." To formally describe type judgment, we also introduce a special type `T` for timestamps. As we do not consider any subclasses, there are no subtyping rules in the type system. All typing rules in Figure 8 are straightforward. We note that T-Set checks that the number of arguments for `set` is same as the number of source signals of the receiver, and the type of each argument matches the type of the corresponding persistent signal.

Typing rules for method and class declarations are shown in Figure 9. A type judgment for methods in a class is of the form `M ok in C`, read as "method `M` is well-formed in class `C`." The typing rule T-Method only checks that the method body is given the declared type $\mathtt{C}_0$ under the type environment constructed by formal parameters and the special variable `this`. A signal class `C` is well-formed if the right-hand side expressions $\overline{\mathtt{e}}$ of all the non-source signals are given the declared type under the type environment `this` : `C`, and all methods are well-formed. Furthermore, it checks that each $\mathtt{e}_i$ in $\overline{\mathtt{e}}$ is side-effect-free. As explained

$$\frac{\mu(\mathtt{l}) = \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mathtt{p} \in \mathcal{R}_{\mathtt{C}(\mathtt{l})}}{\mathit{sources}(\mu, \mathtt{l.p}) = \{\ \mathtt{l.p}\ \}}$$

$$\frac{\mu(\mathtt{l}) = \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mathit{FP}(\mathtt{C}, \mathtt{p}) = \mathtt{e} \qquad \forall \mathtt{e}_i.\mathtt{p}_i \in \mathtt{e}.\mu \mid \mathtt{e}_i.\mathtt{p}_i \longrightarrow^* \mu_i \mid \mathtt{l}_i.\mathtt{p}_i}{\mathit{sources}(\mu, \mathtt{l.p}) = \bigcup_i \mathit{sources}(\mu_i, \mathtt{l}_i.\mathtt{p}_i)}$$

Figure 10 Source signal lookup.

$$\frac{\mu(\mathtt{l}_0) = \mathcal{R}_{\mathtt{C}(\mathtt{l}_0)} \qquad \mathtt{p}_0 \in \mathcal{R}_{\mathtt{C}(\mathtt{l}_0)}}{\mathit{time}_{\mathtt{l}_0.\mathtt{p}_0}(\mu, \mathtt{l}_0.\mathtt{p}_0) = \pi_{\mathit{time}}(\mathcal{R}_{\mathtt{C}(\mathtt{l}_0)})}$$

$$\frac{\mu(\mathtt{l}) = \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mathtt{p} \in \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mathtt{l}_0 \neq \mathtt{l} \vee \mathtt{p}_0 \neq \mathtt{p}}{\mathit{time}_{\mathtt{l}_0.\mathtt{p}_0}(\mu, \mathtt{l.p}) = \emptyset}$$

$$\frac{\mu(\mathtt{l}) = \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mathit{FP}(\mathtt{C}, \mathtt{p}) = \mathtt{e} \qquad \forall \mathtt{e}_i.\mathtt{p}_i \in \mathtt{e}.\mu \mid \mathtt{e}_i.\mathtt{p}_i \longrightarrow^* \mu_i \mid \mathtt{l}_i.\mathtt{p}_i}{\mathit{time}_{\mathtt{l}_0.\mathtt{p}_0}(\mu, \mathtt{l.p}) = \bigcup_i \mathit{time}_{\mathtt{l}_0.\mathtt{p}_0}(\mu_i, \mathtt{l}_i.\mathtt{p}_i)}$$

Figure 11 Source signal's time-series.

earlier, our language does not provide this check but relies on an external checker. Thus, we just define the predicate *sideeffectfree*($\mathtt{e}_0$) as follows:

$$\frac{\forall \mathtt{e} \in \text{subexpressions of } \mathtt{e}_0.(\mu \mid \mathtt{e} \longrightarrow^n \mu \mid \mathtt{l} \text{ for some } \mathtt{l} \wedge \forall i \leq n.\mu \mid \mathtt{e} \longrightarrow^i \mu \mid \mathtt{e}' \text{ for some } \mathtt{e}')}{\mathit{sideeffectfree}(\mathtt{e}_0)}$$

This means that each $\mathtt{e}_i$ does not change the runtime environment during its reduction.

5.3 Properties

To formally state the glitch-freedom in our calculus, we further introduce two other auxiliary definitions that perform signal network traverse. Figure 10 defines the source signal lookup $\mathit{sources}(\mu, \mathtt{l.p})$ that returns a set consisting of all the source signals on which `l.p` depend. If `l.p` is a source, which means that the value of `p` is stored in the relation $\mathcal{R}$ of the receiver, $\mathit{sources}(\mu, \mathtt{l.p})$ just returns the singleton of `l.p`. Otherwise, it recursively searches all the source signals by obtaining all signals contained in the right-hand side expression `e` of `l.p`. Similarly, we define the auxiliary definition $\mathit{time}_{\mathtt{l}_0.\mathtt{p}_0}(\mu, \mathtt{l.p})$ that returns the set of timestamps of the source signal $\mathtt{l}_0.\mathtt{p}_0$ *that is observed from signal* `l.p`. Intuitively, the calculus is glitch-free if two or more subexpressions of the right-hand side of `l.p` depends on the same source signal $\mathtt{l}_0.\mathtt{p}_0$, then the same set of timestamps of $\mathtt{l}_0.\mathtt{p}_0$ is observed from any of those subexpressions. We formally describe this property as the following theorem.

▶ **Theorem 5.1** (glitch-freedom). *Let* `signal class C { ... } ok`, $\mu(\mathtt{l}_0) = \mathcal{R}_{\mathtt{C}(\mathtt{l}_0)}$ *for some* μ, *and* $\mathtt{p}_0$ *is a non-source signal declared in* `C` *whose right-hand side is specified as* $\mathtt{e}_0$. *For all subexpressions* `e.p` *and* $\mathtt{e}'.\mathtt{p}'$ *in* $\mathtt{e}_0$, *we have* $\forall s \in \mathit{sources}(\mu, \mathtt{e.p}) \cap \mathit{sources}(\mu, \mathtt{e}'.\mathtt{p}').\ \mathit{time}_s(\mu, \mathtt{e.p}) = \mathit{time}_s(\mu, \mathtt{e}'.\mathtt{p}')$.

Proof. By `signal class C { ... } ok`, we have *sideeffectfree*($\mathtt{e}_0$). This means that we always access the same μ during the traversals of subexpressions `e.p` and $\mathtt{e}'.\mathtt{p}'$ (i.e., μ does not change in the premises of definitions of *sources*$(\mu, \mathtt{l.p})$ and $time_{\mathtt{l}_0.\mathtt{p}_0}(\mu, \mathtt{l.p})$). Thus, it is obvious that $time_s(\mu, \mathtt{e.p}) = time_s(\mu, \mathtt{e}'.\mathtt{p}')$ for all s in *sources*$(\mu, \mathtt{e.p}) \cap$*sources*$(\mu, \mathtt{e}'.\mathtt{p}')$. ◀

Another remaining issue is the type soundness. Even though our type system is very simple (e.g., there is no subtyping), formulation of the type soundness is not easy as expected, because our calculus interacts with the database system. We first define the judgment $\Sigma \vdash \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$, read "relation $\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$ is well-formed under the environment Σ," which indicates that $\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$ is not empty and all values in $\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$ is well-typed, as follows:

$$\frac{\emptyset \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}_0 \qquad \forall \mathtt{p} \in att(\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}).\emptyset \mid \Sigma \vdash \pi_{\mathtt{p}}(\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}) : \mathtt{C} \wedge signalType(\mathtt{C}_0, \mathtt{p}) = \mathtt{C} \text{ for some } \mathtt{C}}{\Sigma \vdash \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}}$$

In this definition, we write the set of attributes in $\mathcal{R}$ as $att(\mathcal{R})$. The judgment $\Sigma \vdash \pi_{\mathtt{p}}(\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}) : \mathtt{C}$ returns true if all values in $\pi_{\mathtt{p}}(\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)})$ have type `C`. Then, we define the well-formedness of a runtime environment as follows.

▶ **Definition 5.2.** *A runtime environment μ is said to be well-formed with respect to an identifier environment Σ, written $\Sigma \vdash \mu$, if $dom(\mu) = dom(\Sigma)$ and $\Sigma \vdash \mu(\mathtt{l})$ for every $\mathtt{l} \in dom(\mu)$.*

We note that the well-formedness of the runtime environment is not always held during the computation. For example, if the redex has the form `new C(l)`, the runtime environment contains an empty relation after the reduction. Thus, the type preservation theorem is formulated as follows.

▶ **Theorem 5.3** (preservasion). *Suppose that $\forall \mathtt{CL} \in dom(CL).\mathtt{CL}$ `ok`. If $\Gamma \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$, $\Sigma \vdash \mu$, and $\mu \mid \mathtt{e} \longrightarrow \mu' \mid \mathtt{e}'$, then $\Gamma \mid \Sigma' \vdash \mathtt{e}' : \mathtt{C}$ for some $\Sigma' \supseteq \Sigma$, and $\mu' = \mu \oplus \{\mathtt{l} \mapsto \emptyset\}$ or $\Sigma' \vdash \mu'$.*

Proof. See Appendix A.1. ◀

We also need to consider the fact that a database query may fail. For example, the type system cannot prohibit the use of a timestamp that is earlier than the beginning of the computation. This observation results in the following progress theorem.

▶ **Theorem 5.4** (progress). *Suppose that $\emptyset \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$ for some `C` and Σ. Then, either `e` is an identifier or a time-oriented query `l.snapshot(t)` where $\mu(\mathtt{l}) = \sigma_{\mathtt{t} < time}(\mu(\mathtt{l}))$ for some μ and `t`, or, for any μ such that $\Sigma \vdash \mu$, there are some expression $\mathtt{e}'$ such that $\mu \mid \mathtt{e} \longrightarrow \mu' \mid \mathtt{e}'$ where $\mu' = \mu \oplus \{\mathtt{l} \mapsto \emptyset\}$ or $\Sigma' \vdash \mu'$ for some $\Sigma' \supseteq \Sigma$.*

Proof. See Appendix A.2. ◀

One issue for ensuring type soundness is that the runtime environment μ can contain an empty relation during the computation. An access to a signal bound with such a relation definitely fails, and to avoid such an access, the empty relation should be populated before an access to the signal occurs. To address this issue, we simply take an approach where all signal class instances are enforced to be immediately populated using `set` after their creations. This is a simplification of the SignalJ's solution discussed in Section 6.2; i.e., the calculus does not model the lifecycle management but is developed to be applicable to this management. To describe this enforcement, we define the well-formedness of expressions.

▶ **Definition 5.5.** *An expression* $\mathtt{e}_0$ *is said to be well-formed (written* $\mathtt{e}$ *wf) if and only if, for all* $\mathtt{e} \in$ *subexpressions of* $\mathtt{e}_0$ *where* $\mathtt{e} = \mathtt{new\ C(l)}$ *for some* $\mathtt{C}$ *and* $\mathtt{l}$, $\mathtt{e}$ *is a subexpression of* $\mathtt{e.set(\overline{e})}$ *for some* $\overline{\mathtt{e}}$ *wf.*

Using this definition, we define the well-formedness of method and class declarations as follows.

$$\frac{\mathtt{e}\ \mathit{wf}}{\mathtt{C\ m(\overline{C}\ \overline{x})\ \{\ return\ e;\ \}}\ \mathit{wf}}$$

$$\frac{\overline{\mathtt{e}}\ \mathit{wf} \qquad \overline{\mathtt{M}}\ \mathit{wf}}{\mathtt{signal\ class\ C\ \{\ ...;\ signal\ \overline{C}\ \overline{p}{=}\overline{e};\ \overline{M}\ \}}\ \mathit{wf}}$$

We write CT *wf* if all class declarations in CT are well-formed. Then, our type soundness theorem is formulated as follows.

▶ **Theorem 5.6** (type soundness)**.** *Consider a program* $(CT, \mathtt{e})$ *with* CT *wf,* $\emptyset \mid \emptyset \vdash \mathtt{e} : \mathtt{C}$*, and* $\mathtt{e}$ *wf. If* $\emptyset \mid \mathtt{e} \longrightarrow^* \mu \mid \mathtt{e}'$ *for some* μ *with* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *is either an identifier* $\mathtt{l}$ *with* $\emptyset \mid \Sigma \vdash \mathtt{l} : \mathtt{C}$ *for some* Σ*, or an expression containing a time-oriented query* $\mathtt{l.snapshot(t)}$ *where* $\mu(\mathtt{l}) = \sigma_{\mathtt{t}<time}(\mu(\mathtt{l}))$ *for some* μ *and* $\mathtt{t}$*.*

Proof. By induction on the length of $\emptyset \mid \mathtt{e} \longrightarrow^* \mu \mid \mathtt{e}'$. If $\{\mathtt{l} \mapsto \emptyset\} \in \mu'$ where $\emptyset \mid \mathtt{e} \longrightarrow^* \mu' \mid \mathtt{e}''$ for some $\mathtt{e}''$, it is easy to show that the last applied computation rule is R-New. As $\mathtt{e}$ *wf* and CT *wf*, all `new` expressions in $\mathtt{e}$ and CT are qualified by a `set` call, and because of the evaluation order defined by R-Cngl, the following reduction always use R-Set, resulting in the reduction $\mu' \mid \mathtt{e}'' \longrightarrow_{\text{R-Set}} \mu'' \mid \mathtt{e}'''$ and $\Sigma \vdash \mu''$ for some Σ. Then, Theorems 5.3 and 5.4 finishes the case. Other cases are straightforward. ◀

6 Implementation

The proposed mechanism is implemented on TimescaleDB. We show the runtime architecture of signal class instances in Figure 12. A signal class is compiled into a normal Java class. Each compiled Java class uses the runtime library that prepares the connections to the underlying database system and implements the runtime semantics of persistent signals.

6.1 Compilation

The compiler is implemented using ExtendJ [9]. Figure 12 shows the object diagram after the compilation, where a signal class is translated into a Java class that implements the `SignalClassInstance` interface, which provides the methods necessary for signal class lifecycle management. The implementations of those methods are automatically inserted into the class during the compilation.

Each persistent signal is converted into an instance of `PersistentSignal`, which is a part of the runtime library. Each `PersistentSignal` instance encapsulates the database table that contains all updates of the persistent signal. Every access to the persistent signal is rewritten to the method invocation that returns the "current value" of that signal, and every imperative operation that changes the value of the persistent signal (e.g., a reassignment using `=`) is converted into the method invocation that updates the underlying database table. More precisely, this update is not immediately performed when the reassignment on the persistent signal is issued; it is postponed until the update requests on all the persistent

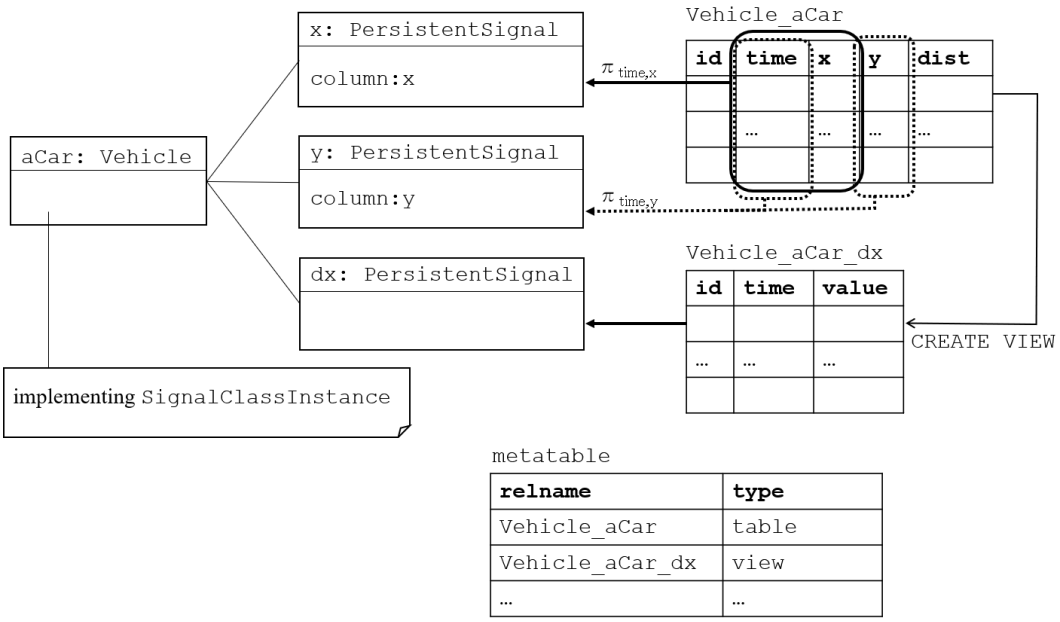


Figure 12 Runtime architecture of signal class instances. Every signal class is introduced with the `SignalClassInstance` interface by the compiler. Each persistent and view signal is converted to an instance of `PersistentSignal`, which is a part of the runtime library that implements the API methods. This instance accesses the projection of the underlying database table and views. There is also the `metatable` that manages the existing signal class instances.

signals contained in the signal class instance are issued. An instance of `Synchronizer`, which is also a part of the runtime library attached to the signal class instance, monitors all the persistent signals in the signal class instance and calls the `set` method that updates the underlying database table according to the provided synchronization policy such as non-blocking-buffered, non-blocking-bufferless, blocking, and asynchronous.

Each constructor of the signal class is also translated to create all `PersistentSignal` instances declared in the signal class. When the instance of `PersistentSignal` is created, it is tested whether the corresponding database table already exists; if so, the `PersistentSignal` instance is connected with that table; if not, a new table is created. Similarly, every view signal, which is also an instance of `PersistentSignal`, is created by calling the API method that is prepared in the runtime library in advance. For the creation of these `PersistentSignal` instances, the compiler simply inserts a piece of code that calls these runtime library methods. As there are chains of dependency between persistent and view signals, these creations of `PersistentSignal` instances are topologically sorted in a similar manner to those in Flapjax [20]. An instance of `Synchronizer` is also created within the constructor execution that monitors all updates of `PersistentSignal` instances.

6.2 Naming and initialization

As explained in Section 4.2, the identifier of a signal class instance is always provided when it is created. In Figure 1, the `id` parameter in the constructor of `Vehicle` is mandatory. Internally, this identifier is used to determine the names of tables and views. The name of persistent signal table is determined by the fully-qualified name of signal class and the identifier. The name of the view is determined by the name of the table and the name of the

view signal. For example, consider the following creation of the `Vehicle` instance again:

```
Vehicle aCar = new Vehicle("501a1234", "Haskell", "Toyota", "Sienta");
```

Assuming that `Vehicle` is declared in the `vehicletracking` package, the names of the table and views are determined as follows.

```
vehicletracking_Vehicle_Oita501a1234      // table for persistent signals
vehicletracking_Vehicle_Oita501a1234_x12h // view for x12h
vehicletracking_Vehicle_Oita501a1234_y12h // view for y12h
vehicletracking_Vehicle_Oita501a1234_dx   // view for dx
vehicletracking_Vehicle_Oita501a1234_dy   // view for dy
vehicletracking_Vehicle_Oita501a1234_v    // view for v
```

The signal class instance encapsulates these table and views. We cannot create multiple `Vehicle` instances with the same identifier, but we can still use the same identifier in the instances of other signal classes.

One subtle issue is providing an initial value to a persistent signal. Theorem 5.6 indicates that the program sticks only when an unexpected timestamp is chosen for the time-oriented query *if the program is well-formed*. This definition of well-formedness requires that a signal class instance should be immediately initialized using `set`. However, in the real program there is also a situation where the instance is bound with an existing database table. In such a case, the call of `set` should not incur any effects.

To ensure that the initialization is performed only when the history of the persistent signal is empty, the `SignalClassInstance` interface in Figure 3 provides an additional method:

```
  public void setIfNotInitialized(Object ... newValues);
```

This method sets the values provided as arguments to persistent signals declared in the receiver signal class instance only if their histories are empty. We note that, like `set`, this interface is defined for the runtime library.

We note that the call of `reset` also makes the execution history empty, and currently our compiler does not check the well-formedness of the program. Instead, the runtime system raises an exception when an empty execution history is accessed.

6.3 Database implementation

TimescaleDB is an open-source time-series database that can run at edge systems as well as in the cloud. Thus, we can implement a variety of applications, including an IoT system where the time-series data is managed in an edge system and a data center that manages massive amount of time-series data. As it is a relational database, the implementation of the dynamic semantics in Section 5, which is based on the relational algebra, is straightforward.

All `PersistentSignal` instances in a signal class instance are connected with the underlying database system when it is created. They access the table for persistent signals and views that corresponds to view signals. Those table and views are created if they do not exist (i.e., if the **new** event is fired with the initial state in Figure 2). The view creating API in Table 1 is also applicable in our system. For example, the expression "`x.within(ts, "12 hours")`", where `x` is a persistent signal containing the x-coordinate of the running vehicle, executes the following `SELECT` query to create a view (`rel_name` is the table that `x` refers to):

```
SELECT time,x FROM [rel_name] WHERE time > ts - interval '12 hours'
```

Table 2 Performance evaluation results. Every measurement is expressed in millisecond (ms), and was performed by taking an average of 10 vehicles.

(a) Response time of persistent/view signal

# records	`x`	`x12h`	`dx`	`v`
100	0.7	0.9	1.2	1.4
1000	0.8	1.0	1.6	1.9
10000	0.8	1.6	6.2	6.1

(b) Overhead of vehicle creation

table/view creation	average time
w/	48.7
w/o	42.1

Normally, the database system does not provide a mechanism to group those related table and views. Thus, the binding between a signal class instance and its corresponding table and views is maintained using the naming mechanism explained in Section 6.2.

Each table for persistent signals of a signal class instance (let the name of this instance be `a`) consists of tuples of persistent signal values with their timestamps. If the persistent signal refers to another signal class instance, the database table contains the name of that instance. When accessed, that instance is restored from the database: if that signal class instance is active, we obtain that instance from the hashtable that contains all active signal class instances; otherwise, a new signal class instance is created using the name stored in the database. The database table `metatable` remembers the names of signal class instances that have been created to date, including inactive ones.

6.4 Performance Evaluation

To confirm that the explained application scenario is realistic in the proposed implementation, we performed simple microbenchmark experiments that measure the response time of persistent signal accesses. These microbenchmarks were performed using TimescaleDB as a backend, which is running on Linux kernel version 4.18.0. This system was running on six-cores Intel Zeon E-2276G 3.80GHz with 16GB main memory and 512GB SSD. TimescaleDB was tuned to have recommended memory settings, including 2GB shared buffers, 6GB effective cache size, 1GB maintenance working memory, and 26,214KB working memory.

In these microbenchmarks, we first prepared histories of vehicles by virtually running them, and then measured the performance of accesses to signals `x` (holding the x-coordinate of each vehicle), `x12h` (holding the last 12 hours of data of x-coordinate), `dx` (holding the difference between `x12h` and its previous value), and `v` (holding the estimated velocity of the vehicle). Before these signals were accessed, each vehicle's timestamp was randomly set by issuing `snapshot`.

Table 2a summarizes the response time of accesses to persistent and view signals. The performance depends on the amount of records the history has. Accesses to view signals `dx` (calculated using join) and `v` (calculated using embedded functions) require around 6 ms when the history has 10,000 records. To confirm that this result is acceptable, we also implemented a vehicle viewer that displays 10 vehicles with 10,000 records and 7 signals (including the y-coordinate `y`, the last 12 hours of y-coordinate `y12h`, and the difference between `y12h` and its previous value, in addition to signals that are shown in Table 2) of each vehicle. This viewer provides a slider to allow the user time-travel, and 70 signals in total are recalculated at once when a specific timestamp is set using the slider. In this viewer, we observed that the slider was mostly responsive.

Table 2b shows the overhead of vehicle creation. This depends on whether the vehicle instance is created by connecting the existing table and views, or creating new ones (i.e., new `id` is introduced). The creation of table and views (it consists of one table and 5 views)

requires around 6 ms. Other overhead includes making connections to the database system. This overhead looks relatively large, but we can reduce this by sharing the connections to the database.

7 Related Work

Signals are a well-known abstraction in reactive programming (RP), which have been inspired by synchronous languages [12, 4, 29] and functional-reactive programming (FRP) languages [10]. FRP features are now available in general-purpose functional languages (e.g., the Yampa library [24] is available for Haskell), and recently they have made their way into imperative object-oriented settings [20, 30, 17] by integrating signals with event-based programming features (such as the event mechanism proposed by EScala [11]).

Even though Yampa's switch and our switching mechanism look somewhat alike, there are fundamental differences between them. First, in our switching, the old sub-network (e.g., the monitored vehicle) is not lost after switching and can be accessed if its `id` is restored. In Yampa, on the other hand, the old signal is lost and we need to preserve every measure manually if we want to access that again. Second, in our switching, there is no guarantee that the switching is performed at the same time when the vehicle is updated, while in Yampa, switches always occur at a global time step. In short, signal classes provide a more general switching with less guarantees.

Although signals in RP languages are not persistent, some research efforts have been made to record the update histories of signals to make them available for debugging. For example, time-traveling [25] makes it possible to pause the execution and rewind to any earlier execution point. This technique is now common in RP debuggers. Reactive Inspector [31], a debugger for REScala [30], visualizes how signal networks are constructed and evolved and how propagations take place over those networks during execution. Using this debugger, a programmer can see the status of the networks at any execution point. Another way of debugging FRP programs is to use *temporal propositions*, an FRP construct based on linear temporal logic [27]. Time-traveling in FRP can also been seen in the literature [28] that presents a uniform way to control how time flows, such as the direction of time flow and sampling rate, by giving time transformations over time domains. Some tools also provide visualization of such time-series data, such as allowing viewing of the execution history in a single display to identify anomaly propagation patterns that are repeated over time [3, 14, 13]. Usually, such tools are dedicated to debugging; thus, they record the history of one execution. Persistence across multiple executions, such as that discussed in the proposed lifecycle model, is not considered. Furthermore, time-series data handled in such tools are not provided for use by applications. For example, no convenient APIs to query over such time-series data are provided.

Other research efforts that are relevant to persistent signals include fault-tolerant RP [21, 22] that provides an implementation for snapshotting mechanism of signals. In contrast, SignalJ focuses more on applications that query over time-series data, which is also evident in the formalization developed using relational algebra. In a larger picture, such time-series data can be open, i.e., that are shared with and queried from multiple processes by referring to that using the identifiers.

As discussed in the implementation of our system, time-series databases provide important techniques to implement persistent signals. Jensen et al. presented a survey on time-series databases, which are also known as time-series management systems [16]. In their survey, time-series databases were categorized as internal data stores, external data stores, and relational

database extensions. An internal data store (e.g., *tsdb* presented by Deri et al. [8]) integrates both a data store and a processing engine together in the same application, allowing for deep integration between the storage and processing engine. In another approach, an external data store (e.g., *Gorilla* by Pelkonen et al. [26] and *BTrDB* by Andersen and Culler [1]) uses an existing external management system, allowing for existing system deployments to be reused. Finally, an relational database extension (e.g., *TimeTravel* by Khalefa et al. [19]) allows the expressive power of the relational database to be applied to the time-series database. TimescaleDB, as used in our implementation, falls into this last category. Overall, there have been many time-series database implementations suitable for different use case scenarios. Therefore, although we consider that the performance of TimescaleDB is satisfactory in many cases, it will be beneficial to consider other implementations that might be suitable for some specific application domain.

Finally, we do not consider the proposed persistent signal lifecyle model as new because there have been much work on persistent objects where the lifetime of the objects can be indefinite (e.g., [2]). We keep the model as simple as possible to extend it to the objects containing a set of time-varying values. There have also been much work on the implementation of persistent objects using SQL (e.g., [7]). Instead, in our system, the mapping is defined only for the trivial cases, i.e., the mapping from persistent signals to the table. We do not define the mapping for view signals; it is left for the programmers or domain engineers. However, we consider some of this definition could be performed automatically using program synthesis. Actually, program synthesis for SQL queries has recently been intensively studied (e.g., the work by [32]). We consider application of such technologies to automatic synthesis of view signals is also an interesting direction for future work.

8 Concluding Remarks

In this paper, we proposed a new language mechanism signal class, which encapsulates a network of related persistent and view signals. Not only does this mechanism allow us to represent persistent time-varying values with complex data types, but it also provides a unit of lifecycle management and a unit of synchronization. All these features overcome the drawbacks of existing persistent signals in that they cannot represent persistent time-varying values with complex data types, they must be created only at compile time, and the network is connected only using pre-defined methods. We clarified how each signal class instance behaves by defining its lifecycle model and formal semantics that maps each signal class instance to the underlying database system using relational algebra. In these definitions, we confirmed several properties regarding database transparency, glitch-freedom, and type soundness. All these results indicate that our approach is effective to implement reactive systems using convenient abstractions of time-varying values with their execution histories.

References

1 Michael P. Andersen and David E. Culler. BTrDB: Optimizing storage system design for timeseries processing. In *14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 39–52, 2016.

2 M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *SIGMOD Record*, 25(4):68–75, 1996.

3 Herman Banken, Erik Meijer, and Georgious Gousios. Debugging data flows in reactive programs. In *ICSE'18*, pages 752–763, 2018.

4 Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. doi:10.1016/0167-6423(92)90005-V.

5 Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

6 Gregory H. Cooper. *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language*. PhD thesis, Department of Computer Science, Brown University, 2008.

7 S. Dar, N.H̃. Gehani, and H.Ṽ. Jagadish. CQL++: A SQL for the Ode object-oriented DBMS. In *Advances in Database Technology — EDBT '92*, volume 580 of *LNCS*, pages 201–216, 1992.

8 Luca Deri, Simone Mainardi, and Francesco Fusco. tsdb: A compressed database for time series. In *TMA*, 2012.

9 Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'07)*, pages 1–18, 2007. doi:10.1145/1297105.1297029.

10 Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 263–273, 1997. doi:10.1145/258949.258973.

11 Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: modular event-driven object interactions in Scala. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD'11)*, pages 227–240, 2011. doi:10.1145/1960275.1960303.

12 Nicholas Halbwachs, Paul Caspi, Pascal Paymond, and Daniel Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991. doi:10.1109/5.97300.

13 Takumi Hikosaka, Tetsuo Kamina, and Katsuhisa Maruyama. Visualizing reactive execution history using propagation traces. In *REBLS'18*, 2018.

14 Jeff Horemans and Bob Reynders. Elmsvuur: A multi-tier version of elm and its time-traveling debugger. In *TFP 2017*, volume 10788 of *LNCS*, pages 79–97, 2017.

15 Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

16 Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. Time series management systems: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 29:2581–2600, 2018.

17 Tetsuo Kamina and Tomoyuki Aotani. Harmonizing signals and events with a lightweight extension to Java. *The Art, Science, and Engineering of Programming*, 2(3), 2018. doi:10.22152/programming-journal.org/2018/2/5.

18 Tetsuo Kamina and Tomoyuki Aotani. An approach for persistent time-varying values. In *Onward!'19*, pages 17–31, 2019.

19 Mohamed E. Khalefa, Ulrike Fischer, Torben Bach Pedersen, and Wolfgang Lehner. Model-based integration of past & future in TimeTravel. In *Proceedings of the VLDB Endowment (PVLDB)*, pages 1974–1977, 2012.

20 Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA'09)*, pages 1–20, 2009. doi:10.1145/1640089.1640091.

21 Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. Fault-tolerant distributed reactive programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:26, 2018.

22 Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. A fault-tolerant programming model for distributed interactive applications. *Proc. ACM Program. Lang.*, 3(OOPSLA):144:1–144:29, 2019.

23 Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. Distributed reactive programming for reactive distributed systems. *The Art, Science, and Engineering of Programming*, 3(3):5:1–5:52, 2019.

24 Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell'02)*, pages 51–64, 2002. doi:10.1145/581690.581695.

25 Laszlo Pandy. Bret Victor style reactive debugging. Elm Workshop, 2013.

26 Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proc. VLDB Endow.*, 8(12):1816–1827, 2015.

27 Ivan Perez. Back to the future: time travel in FRP. In *Haskell'17*, pages 105–116, 2017.

28 Ivan Perez and Henrik Nilsson. Testing and debugging functional reactive programming. *Proceedings of the ACM on Programming Languages*, 1, 2017.

29 Marc Pouzet. *Lucid Synchrone version 3.0: Tutorial and Reference Manual*. Université Paris-Sud, LRI, April 2006. Online manual. URL: https://www.di.ens.fr/~pouzet/lucid-synchrone/lucid-synchrone-3.0-manual.pdf.

30 Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY'14)*, pages 25–36, 2014. doi:10.1145/2577080.2577083.

31 Guido Salvaneschi and Mira Mezini. Debugging for reactive programming. In *ICSE'16*, pages 796–807, 2016.

32 Sai Zhang and Yuyin Sun. Automatically synthesizing SQL queries from input-output examples. In *ASE'13*, pages 224–234, 2013.

33 YungYu Zhuang. A lightweight push-pull mechanism for implicitly using signals in imperative programming. *Journal of Computer Languages*, 54, 2019.

A Proofs

A.1 Proof of Theorem 5.3

We first show some lemmas required by the proof of Theorem 5.3.

▶ **Lemma A.1** (weakening).
1. *If* $\Gamma \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$ *and* $\mathtt{x} \notin \Gamma$, *then* $\Gamma, \mathtt{x} : \mathtt{D} \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$.
2. *If* $\Gamma \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$ *and* $\mathtt{l} \notin \Sigma$, *then* $\Gamma \mid \Sigma, \mathtt{l} : \mathtt{C} \vdash \mathtt{e} : \mathtt{C}$.

Proof. By straightforward induction on $\Gamma \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$. ◀

▶ **Lemma A.2** (substitution). *If* $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{C}} \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0$ *and* $\Gamma \mid \Sigma \vdash \overline{\mathtt{l}} : \overline{\mathtt{C}}$, *then* $\Gamma \mid \Sigma \vdash [\overline{\mathtt{l}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{C}_0$.

Proof. By induction on $\Gamma \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$. ◀

Proof of Theorem 5.3. By induction on the derivation of $\mu \mid \mathtt{e} \longrightarrow \mu' \mid \mathtt{e}'$.

Case R-PSIGNAL: $\mathtt{e} = \mathtt{l}_0\mathtt{.p} \quad \mathtt{e}' = \mathtt{l} \quad \mu(\mathtt{l}_0) = \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)} \quad \pi_{\mathtt{p}}(\sigma_{latest}(\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)})) = \mathtt{l}$

By T-SIGNAL, $\Gamma \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}'_0$ and $signalType(\mathtt{C}'_0, \mathtt{p}) = \mathtt{C}$ for some $\mathtt{C}'_0$. As $\Sigma \vdash \mu$, we have $\Sigma \vdash \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$, and by the definition of $\Sigma \vdash \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$, we have $\emptyset \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}_0$. By Lemma A.1, $\Gamma \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}_0$. Thus $\mathtt{C}'_0 = \mathtt{C}_0$. By the definition of $\Sigma \vdash \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$ and T-SIGNAL, we have $\pi_{\mathtt{p}}(\sigma_{latest}(\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}))$. Then, Lemma A.1 finishes the case.

Case R-VSIGNAL: $\mathtt{e} = \mathtt{l}_0\mathtt{.p} \quad \mathtt{e}' = \mathtt{e}_0 \quad \mu(\mathtt{l}_0) = \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)} \quad signalExpr(\mathtt{C}_0, \mathtt{p}) = \mathtt{e}_0$

By T-SIGNAL, $\Gamma \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}'_0$ and $\mathit{signalType}(\mathtt{C}'_0, \mathtt{p}) = \mathtt{C}$ for some $\mathtt{C}'_0$. As $\Sigma \vdash \mu$, we have $\Sigma \vdash \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$, and by the definition of $\Sigma \vdash \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$, we have $\emptyset \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}_0$. By Lemma A.1, $\Gamma \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}_0$. Thus $\mathtt{C}'_0 = \mathtt{C}_0$. By T-CLASS and the definitions of *signalExpr* and *signalType*, we have $\mathtt{this} : \mathtt{C}_0 \mid \emptyset \vdash \mathtt{e}_0 : \mathtt{C}$. Then, Lemma A.1 finishes the case.

Case R-NEW: $\mathtt{e} = \mathtt{new\ C(l)} \quad \mathtt{e}' = \mathtt{l} \quad \mu' = \mu \oplus \{\mathtt{l} \mapsto \emptyset\}$

Let $\Sigma' = \Sigma, \mathtt{l} : \mathtt{C}$. By T-ID, $\Sigma' \vdash \mathtt{l} : \mathtt{C}$, finishing the case.

Case R-SET: $\mathtt{e} = \mathtt{l.set}(\overline{\mathtt{l}}) \quad \mathtt{e}' = \mathtt{l}$

It is easy to show that $\Gamma \mid \Sigma \vdash \mu'$, and by T-SET we have $\Gamma \mid \Sigma \vdash \mathtt{l} : \mathtt{C}$, finishing the case.

Case R-TIME: $\mathtt{e} = \mathtt{l}_0\mathtt{.snapshot(t)} \quad \mathtt{e}' = \mathtt{l}_0$

It is easy to show that $\Gamma \mid \Sigma \vdash \mu'$, and by T-TIME we have $\Gamma \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}$, finishing the case.

Case R-INVK: Finished by Lemma A.2, T-INVK, and definitions of *mtype* and *mbody*.

Case R-CNGL: Finished by the induction hypothesis. ◀

A.2 Proof of Theorem 5.4

Proof of Theorem 5.4. By induction on the derivation of $\Gamma \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$.

Cases T-VAR and T-TS: Cannot occur.

Case T-ID: Immediately finished.

Case T-SIGNAL: $\mathtt{e} = \mathtt{e}_0\mathtt{.p} \quad \emptyset \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \quad \mathit{signalType}(\mathtt{C}_0, \mathtt{p}) = \mathtt{C}$

There are three subcases based on the form of $\mathtt{e}_0$:

Subcase 1: $\mathtt{e}_0 = \mathtt{l}_0$

There are further subcases based on the definition of *signalType*: (1) `signal class C {... persistent signal` $\overline{\mathtt{C}}$ $\overline{\mathtt{p}}$`; ...}` and $\mathtt{p} \in \overline{\mathtt{p}}$. Assuming $\emptyset \mid \Sigma \vdash \mu$, we have $\Sigma \vdash \mu(\mathtt{l}_0)$, i.e., we have a non-empty $\mu(\mathtt{l}_0)$. Thus, $\pi_{\mathtt{p}}(\sigma_{\mathit{latest}}(\mu(\mathtt{l}_0)) = \mathtt{l}$ for some $\mathtt{l}$. Thus, R-PSIGNAL can be applied to $\mathtt{e}$, finishing the case; (2) `signal class C {... signal` $\overline{\mathtt{C}}$ $\overline{\mathtt{p}}$`=`$\overline{\mathtt{e}}$`; ...}` and $\mathtt{p} \in \overline{\mathtt{p}}$. Similarly, R-VSIGNAL finishes the case.

Subcase 2: $\mathtt{e}_0 = \mathtt{l}_0\mathtt{.snapshot(t)}$

Immediately finished because this is the case where $\mathtt{e}$ is an expression containing a time-oriented query.

Subcase 3: Otherwise, R-CONG finishes the case.

Case T-SET: $\mathtt{e} = \mathtt{e}_0\mathtt{.set}(\overline{\mathtt{e}}) \quad \emptyset \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C} \quad \emptyset \mid \Sigma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}}$

There are subcases based on the form of $\mathtt{e}_0$:

Subcase 1: $\mathtt{e}_0 = \mathtt{l}_0$

There are further subcases based on the form of $\overline{\mathtt{e}}$: (1) $\overline{\mathtt{e}} = \overline{\mathtt{l}}$. By T-ID, $\overline{\mathtt{l}} \in \mathit{dom}(\Sigma)$, and assuming $\Sigma \vdash \mu$, we have $\overline{\mathtt{l}} \in \mathit{dom}(\mu)$. We can choose $\mathtt{t} \in \mathit{Time}$ such that $\mathtt{t} > \sigma_{\mathit{latest}}(\pi_{\mathit{time}}(\mu(\mathtt{l}_0)))$. Let $\mathcal{R}'_{\mathtt{C}(\mathtt{l}_0)} = \{(\mathtt{t}, \overline{\mathtt{l}})\} \cup \mu(\mathtt{l}_0)$ and $\mu' = \mu \oplus (\mathtt{l}_0 \mapsto \mathcal{R}'_{\mathtt{C}(\mathtt{l}_0)})$. Then, R-SET finishes the case; (2) Otherwise, R-CONG finishes the case.

Subcase 2: Otherwise, R-CONG finishes the case.

Case T-TIME: $\mathtt{e} = \mathtt{e}_0\mathtt{.snapshot(t)} \quad \emptyset \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C} \quad \emptyset \mid \Sigma \vdash \mathtt{t} : \mathtt{T}$

There are subcases based on the form of $\mathtt{e}_0$:

Subcase 1: $\mathtt{e}_0 = \mathtt{l}_0$

Immediately finished because this is the case where $\mathtt{e}$ is an expression containing a time-oriented query.

Subcase 2: Otherwise, R-CONG finishes the case.

Case T-INVK: $\mathtt{e} = \mathtt{e}_0\mathtt{.m}(\overline{\mathtt{e}}) \quad \emptyset \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \quad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{C}} \to \mathtt{C} \quad \emptyset \mid \Sigma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}}$

There are subcases based on the form of $\mathtt{e}_0$:

Subcase 1: $\mathtt{e}_0 = \mathtt{l}_0$

There are further subcases based on the form of $\overline{\mathtt{e}}$: (1) $\overline{\mathtt{e}} = \overline{\mathtt{l}}$. By the definition of *mtype* and *mbody*, we have $mbody(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{x}}.\mathtt{e}$ where the number of $\overline{\mathtt{l}}$ and that of $\overline{\mathtt{x}}$ are the same. Thus, R-Invk finishes the case; (2) Otherwise, R-Cong finishes the case.

Subcase 2: Otherwise, R-Cong finishes the case.

Case T-New: Immediately finished. ◀

Refinements of Futures Past: Higher-Order Specification with Implicit Refinement Types

Anish Tondwalkar
University of California, San Diego, CA, USA

Matthew Kolosick
University of California, San Diego, CA, USA

Ranjit Jhala
University of California, San Diego, CA, USA

Abstract

Refinement types decorate types with assertions that enable automatic verification. Like assertions, refinements are limited to binders that are in scope, and hence, cannot express higher-order specifications. *Ghost* variables circumvent this limitation but are prohibitively tedious to use as the programmer must divine and explicate their values at all call-sites. We introduce *Implicit Refinement Types* which turn ghost variables into implicit pair and function types, in a way that lets the refinement typechecker automatically *synthesize* their values at compile time. Implicit Refinement Types further take advantage of refinement type information, allowing them to be used as a lightweight verification tool, rather than merely as a technique to automate programming tasks. We evaluate the utility of Implicit Refinement Types by showing how they enable the modular specification and automatic verification of various higher-order examples including stateful protocols, access control, and resource usage.

2012 ACM Subject Classification Theory of computation → Program constructs; Theory of computation → Program specifications; Theory of computation → Program verification

Keywords and phrases Refinement Types, Implicit Parameters, Verification, Dependent Pairs

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.18

Related Version *Extended Version*: `arXiv:2105.01954` [40]

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.3`
archived at `swh:1:snp:aeaf3dbb58f5be84b565e73b5ade1503ee8cb6d6`

Funding This work was supported by NSF grant CCF-1911213.

1 Introduction

Refinement types allow programmers to decorate types with statically checked assertions ("refinements") that can be used for automatic verification over decidable theories, with applications including checking array bounds [48, 34], totality [44], data structure invariants [22], cryptographic protocols [4, 17], and properties of web applications [18].

Problem: Higher-Order Reasoning. Unfortunately, refinements cannot express the *higher-order* specifications needed for higher-order imperative programs [42]. Consider the access-control API:

```
grant :: File → IO ()                    read :: File → IO String
```

Here, `grant` and `read` represent a file access API that enforces access control policies: to `read` a given file, we must have been `grant`'d permission to that file. Concretely, this means that calls to `grant f` update the state of the world to one in which `f` has been added to the set of files we have permission to `read`.

Next, consider the API for operating over a stream of tokens in a linear fashion:

```
init :: ST Token                    next :: Token → ST Token
```

Here, `init` simply gives us the first token to start a stream. To use `next` to recieve a token we must pass it the previous token, which it will then invalidate. Both of these APIs include functions where correct usage fundamentally depends on the state of the world at their eventual call site, as they are composed using higher-order combinators such as `(>>=)` and `(>>)`.

One can formalize these informal specifications by augmenting the source program with *ghost variables* [31, 41, 42] that represent "departed quantities" that characterize the state of the world. In our file access example, `read f` would have to additionally take in the set of files we have been `grant`ed access to, and `f` must be a member, thus allowing us to compose the computations `grant f >> read f`. In our stream example, `next t` must both take in an additional mapping of tokens to their validity, in which `t` must be valid, and then give us back an updated mapping, reflecting that `next t` is the next valid token, thus allowing us to sequence the computations as `next t >>= next`, but not `next t >> next t`. The key idea in both of these examples is that `(>>)` ("sequence") and `(>>=)` ("bind") must *relate* the output of a higher-order argument (`grant f` and `next t`) to the requirements of their other argument (`read f` and `next`). For these stateful combinators, ghost variables allow us to lift the state of the world to relate the output of one computation to the input of another.

Note that while we start with these stateful examples for familiarity, we are interested in the more general problem of reasoning with higher-order programs. For instance, Handley et al. [19] introduce a `Tick` datatype that tracks resource usage of a computation. In this setting, one may want a higher-order constant-resource combinator such as

```
mapA :: (a → Tick n b) → xs:List a → Tick (n * len xs) (List b)
```

Informally, `mapA` maps a computation that uses a fixed amount of resources n (say, time) over a list and guarantees that it will take a fixed amount of resources equal to n times the length of the list. We can use ghost variables to lift the parameter `n` such that both the input function and the output computation can refer to it.

Ghosts of Past and Future. Notice that there is a key distinction between the ghost variables in our informal specifications. The ghost variable for `read` summarizes the history of accesses granted and therefore corresponds to a *history variable* [31] used to refine the past. It is determined *externally* by what accesses have been granted by the time `read` is called. Similarly, `mapA` uses the ghost variable `n` to track costs that will have been incurred once the computation finishes. In contrast, `next t` yields the next valid token, and so the ghost variable for `next` captures the *future* value when the computation is run. This corresponds to a *prophecy variable* [2] we can use to refine the future. That is, `next` makes an *internal* choice about what the next token *will be*. While external choice can be encoded as an additional parameter to `read`, internal choice can be encoded as adding an additional *return* value to `next`. However, in both cases, these ghost variables require polluting code with specification-level details and break APIs that don't accept or produce ghost variables.

Implicit Refinement Types. In this paper, we introduce *Implicit Refinement Types* (IRT), a feature that allows us to capture the above specifications with implicit ghost variables, while preserving the automatic verification properties of prior refinement type systems. Our approach takes inspiration from *implicit parameters* [25, 11, 12], a popular language feature that is the foundation of the theory of typeclasses [46]; models objects [11]; lends flexibility to module systems [47]; and features in C# [16], C++ [20], and Scala. To capture the distinction between internal and external choice of ghost parameters, we introduce two different notions

of implicits: implicit dependent functions (*i.e.* implicit-Π) and implicit dependent pairs (*i.e.* implicit-Σ). Implicit functions capture the notion of *external choice*, where the ghost parameter is determined by the caller of a function. Dually, implicit pairs correspond to *internal choice*, where the ghost parameter is determined by the implementation of the function. To sum up, we make the following contributions:

- A declarative semantics for ***implicit refinement types*** (§3), and an inference algorithm that is sound with respect to the declarative semantics (§5). Crucially, our semantics preserve *subtyping* information, allowing us to take advantage of refinement information in inferring implicit parameters.
- A notion of ***implicit pair types*** that, when added to implicit function types, allows us to express higher-order specifications without requiring any changes to code (§2).
- We implement our system, in a tool dubbed Mist, and ***evaluate*** our prototype on a range of case studies to demonstrate where automatic verification with Implicit Refinement Types bears fruit (§7).

2 Overview

We start with an example-driven overview that walks through specification and verification with implicit functions and pairs on minimal examples of higher-order functions, and then scales these examples to verify the access control, token stream, and resource accounting examples from §1.

2.1 Implicit Function Types

While plain refinement type systems can employ extra parameters to capture ghost variables, they must do so explicitly, requiring programmers to divine their values and implement bookkeeping. To illustrate this, consider the following example of a higher-order function `foo` that accepts a function parameter:

```
foo :: (Bool → Int) → ()
foo f = assert (f True == f False)
```

The static `assert`ion requires that the function passed to `foo` be constant. We could formalize this informal specification by adding an extra ghost parameter to `foo` to represent the singleton return value of its argument:

```
foo :: n:Int → (Bool → SInt n) → ()
```

But now we must not only rewrite `foo` to take this "unused" ghost parameter (`foo n f = assert (f True == f False)`) but must also manually modify the call site to `foo 1 (\z → 1)`. The *implicit function* type lets us handle this automatically. Instead of changing the definition of `foo` and its uses, the programmer changes `foo` to make `n` an *implicit parameter* by surrounding it with square brackets:

```
foo :: [n:Int] → (Bool → SInt n) → ()
```

Verifying programs with implicits. To verify `foo` and its call sites we must check: (1) that the `assert`ion in the body of `foo` is valid (2) that at the call-site `foo (\z → 1)`, the argument meets the precondition. We do so via a bidirectional traversal that checks terms against types to generate and solve *Existential Horn Constraints* (EHC). When checking *inside* the body of `foo`, the implicit parameter `[n:Int]` behaves just like a standard explicit, or "corporeal", parameter one might see in a Π-binder. That is, calls to `foo` will implicitly "pass" an argument

for `n`, so the sequel must hold for *all* values of `n`. This constrains the explicit argument `f` to always return `n`, from which the `assert`ion then follows directly. Checking the *call-site* is more interesting: the implicit parameter `[n:Int]` says the application is valid iff there *exists* a witness `n` such that the remainder of the specification holds. For `foo (\z → 1)`, we require that `n` equals the return value of `(\z → 1)`, *i.e.* `n = 1`. Let's see how to automatically find such witnesses.

Step 1: Templates. First, we generate *templates* to represent the types of terms whose refinements must be inferred. These templates are the base unrefined types for those terms, refined with *predicate variables* κ that represent the unknown refinements [34]. For example, we represent the as-yet unknown type of `\z → 1` with the template $z : \texttt{Bool} \rightarrow \{\nu : \texttt{Int} \mid \kappa(\nu)\}$.

Step 2: Existential Horn Constraints. Next, we traverse the term `foo (\z → 1)` in a bidirectional, syntax directed fashion (§5) to generate the EHC in Figure 1a.

- Constraint (1) comes from the body of the λ-term `\z → 1` and says the type of the returned value `1`, *i.e.* $\{\nu : \texttt{Int} \mid \nu = 1\}$, must be a subtype of the output type $\{\nu : \texttt{Int} \mid \kappa(\nu)\}$.
- Constraint (2) comes from applying `foo` to the λ-term: the type of which must be a subtype of `foo`'s input. Since function outputs are covariant, this means the output type $\{\nu : \texttt{Int} \mid \kappa(\nu)\}$ must be a subtype of $\{\nu : \texttt{Int} \mid \nu = n\}$.

Step 3: NNF Constraints. We say that the EHC in Figure 1a is satisfiable iff there exists a *predicate* for κ that, when substituted into the constraint, yields a *valid* first-order formula. To determine satisfiability, we transform the EHC into a new constraint shown in Figure 1b comprising

- An *NNF Constraint* [6], where we replace each existentially quantified n with a universally quantified version bounded by a predicate variable π, as shown in Constraint (Π), and
- An *Inhabitation Constraint* that each π is non-empty, as shown in Constraint (Σ).

We write π instead of κ here solely to differentiate between predicate variables from the original EHC and those produced in this translation. Intuitively, if every n satisfying $\pi(n)$ satisfies the NNF constraint *and* π is non-empty, *i.e.* there exists n such that $\pi(n)$, then we can conclude there exists some n that satisfies the original EHC.

Step 4: Solution. We require an assignment to the variables π, κ that makes the constraint in Figure 1b valid. We first compute the *strongest* valid solution [10] for κ, which yields $\kappa(v) \doteq v = 1$. As we require π to be inhabited (Constraint (Σ)), we assign to π the *weakest* predicate (as detailed in §6) that makes valid the NNF Constraint (Π), *i.e.*, we assign: $\pi(n) \doteq n = 1$. The above assignments for κ and π make the NNF constraint in Figure 1b valid, thus verifying `foo (\z → 1)`.

2.2 Implicit Pair Types

Implicit function parameters (`[n:Int] → —`) represent *external* choice where the implicit's value is resolved at the call site. But external choice alone is insufficient to capture every use of ghost variables. Consider the following example of a function that *returns* a function:

```
bar :: () → (Bool → Int)
bar _ = (\z → 1)
let f = bar () in assert (f True == f False)
```

$$\wedge\ \forall z.\ true \Rightarrow \forall \nu.\ \nu = 1 \Rightarrow \kappa(\nu) \quad (1)$$
$$\wedge\ \exists n.\ \forall \nu.\ \kappa(\nu) \Rightarrow \nu = n \quad (2)$$

(a) Existential Horn Clause.

$$\wedge\ \forall z.\ true \Rightarrow \forall \nu.\ \nu = 1 \Rightarrow \kappa(\nu)$$
$$\wedge \forall n.\ \pi(n) \Rightarrow \forall \nu.\ \kappa(\nu) \Rightarrow \nu = n \quad (\Pi)$$
$$\wedge\ \exists n.\ \pi(n) \quad (\Sigma)$$

(b) Horn Clause and Side Condition.

Figure 1 Verifying the application `foo (\z →1)`.

Unlike `foo`, `bar` instead *returns* a function `f`: we want to verify that the two calls to `f` return the *same* value. This simple example models our token stream API: while in fact `bar` always returns 1 we do not in general know the exact return value of a computation – for instance the token returned by `next` may be retrieved over the network. Suppose, as for `foo`, we use an implicit function argument to type `bar`:

```
bar :: [n:Int] → () → (Bool → SInt n)
```

Unfortunately, with this type we cannot verify the body of `bar`: for *any* external instantiation of `n`, it requires that `bar` will return a function that returns `n`, which is not true! Instead, `bar` is making an *internal* choice (specifically, that `n = 1`). This motivates our notion of *implicit pair types*, (written `[n:Int]. —`), which add a ghost parameter in the *return* position of a function. This lets us specify

```
bar :: () → [n:Int].(Bool → SInt n)
```

To verify that the code is safe using this specification, we are once again left with two tasks: (1) check that the `assert`ion at the use site of `bar` is valid (2) check at the definition site that the body of `bar` meets the specified postcondition. The first task is the easier one, though it requires an additional step that was not needed in verifying the body of `foo`. *Externally*, the type (`[n:Int]. —`) acts as an extra return value that behaves exactly like a standard ("corporeal") dependent pair and is assumed to exist in the body of the `let`. In order to account for this, we use a type-directed elaboration to automatically insert the appropriate "unpacking", giving a name to `n` at the use site:

```
unpack (n, f) = bar () in assert (f True == f False)
```

Verifying the assertion then follows the exact same logic as verifying the body of `foo`, as `f` is constrained to always return a value equal to `n`.

Now, we turn to the second task: *internally*, the type (`[n:Int]. —`) states that there *exists* some "ghost" value `n` that makes the remaining specification valid. Here, `n` names the return value of the function returned by `bar ()`. When checking the body of `bar`, we will need to find an instantiation of `n` and implicitly "pair" or "pack" this value with the type of the returned function `(\z → 1)`. Here, clearly the instantiation is `n = 1`. Note how the process of checking the definition site of `bar` mirrors that of checking the use site `foo (\z → 1)`. In fact, checking the body of `bar` will generate the exact same constraints shown in Figure 1, which will be solved by the same process, producing the same solution, which suffices to verify that `bar` implements its specification.

2.3 State

Next, we show how Implicit Refinement Types allow us to develop a new way of typing stateful computations, represented as higher-order state transformers. The key challenge here is to devise specification mechanisms that can *relate* the state *after* the transformation

```
-- | A Hoare-style State Transformer -------------------------------
data HST p     q      s a = State (s → (s, a))
type SST w_in w_out s a = HST {w|w = w_in} {w|w = w_out} s a

-- | Read and write the state --------------------------------------
get :: [w:s] → SST w w s {v:s|v = w}
get = State (\s → (s, s))

set :: w:s → HST s {v:s|v = w} s ()
set w = State (\_ → (w, ()))

-- | Monadic Interface for HST -------------------------------------
pure  :: [w:s] → x:a → SST w w s a
»=    :: [w1 w2 w3] →
           SST w1 w2 s a → (a → SST w2 w3 s b) → SST w1 w3 s b

-- | Client: Computing a "fresh" Int  ------------------------------
fresh :: [n:Int] → SST n (n + 1) Int n
fresh = do { n ← get; set (n + 1); pure n }
```

Figure 2 Typing Stateful Computations using Implicits.

with the state *before*. For example, to say that some function *increments* a counter, we need a way to say that the value of the counter after the transformation is one greater than the value before. Previous methods do this either by typing the computation with *two-state* predicates (as in YNot [29]) or with a *predicate transformer* that computes the value of the input state in terms of the output [38].

Implicit Refinement Types enable a new way to relate the input and output states while still ensuring that each atomic component of the specification simply refers to a single value. We will see that implicits are the crucial ingredient, allowing us to *name* – and hence, reason about – the output state, much as they did in the simplified `foo` and `bar` functions.

A Hoare-Style State Transformer Monad. First, as shown in Figure 2, we define a state transformer monad indexed by the type of the state `s` and the computation's result `a`. We call this a *Hoare State Monad* as it is also indexed with two *phantom* parameters `p` and `q` which will be refinement types describing the *input* and *output* states of the transformer. For convenience, we also define the singleton version `SST i o s a` where the `pre`-condition and `post`-conditions are singleton types that say that the input (resp. output) state is exactly `i` (resp. `o`). We can write a combinator to `get` the "current state", represented by the implicit parameter `w` that is both the input and output state, and also used in the singleton type of the result of the computation. Finally, we can write a combinator to `set` the state to some new (explicitly passed) value `w`, in which case, the input state can be any `s`.

A Monadic Interface using Implicit States. Next, we develop a monadic interface for programming with `HST`, by implementing the `pure` and `»=` combinators whose types use implicit parameters to relate their input and output states. The `pure` combinator takes an implicit `w` and returns a "pure" computation `SST w w s a` whose result is the input `x` and where the state is *unchanged*, *i.e.* where the input and output states are both `w`. The bind combinator `(»=)` combines transforms from `w1` to `w2`, and from `w2` to `w3` into a single transform

```
-- | Access control policy State Transformer ------------------------
type AC p1 p2 a = SST p1 p2 (Set String) a

-- | Grant or revoke access to a file path --------------------------
grant :: [p:Set String] → f:String → AC p (p ∪ single f) ()
grant f = State (\p → (insert f p, ()))

revoke :: [p:Set String] → f:String → AC p {v|v = p - single f} ()
revoke f = State (\p → (delete f p, ()))

read :: [p:Set String] → {f:String|f ∈ p} → AC p p String
read f = State (\p → (p, "file␣contents"))

-- | How one might safely read a file -------------------------------
main = runST {} (do grant "f.txt"; read "f.txt")

-- | Enabling dynamic access control policies -----------------------
canRead :: [p:Set String] → f:String → AC p p {v:Bool|v = f ∈ p}
canRead f = State (\p → (p, member f p))

safeRead :: [p:Set String] → f:String → AC p p (Maybe String)
safeRead f = do { r ← canRead;
                  if r then Just <$> read f
                       else pure Nothing }
```

Figure 3 Verifying Access Control Policies.

from `w1` to `w3`. Here, one can think of `w1` as a history variable summarizing the state of the world before the transformed computation and `w2` and `w3` as prophecy variables predicting what the respective sub-transformers will compute. The implicit refinement specification then ensures that these ghost variables all align appropriately.

Verifying Clients. We can use our interface to write a specification and implementation of the function `fresh` that "increments" a counter captured by the state parameter. This program `get`s an integer state `n`, and `set`s it to `n + 1`, and then returns `n`. The `do`-notation desugars into the monadic interface shown above in Figure 2. The specification captures the fact that the "counter" is incremented by relating the input and output states via the implicit parameter `n`. Notice that the implicit parameters are doubly crucial: first, they let us relate the input- and output-states, and second, they make programming pleasant by not requiring the programmer tediously spell out the intermediate states.

2.4 Access Control

The refined Hoare State Transformer lets us specify and verify the access control and token stream examples from §1. In Figure 3 we show how we can instantiate it with refinements over the theory of sets to derive a stateful API representing the verified access control primitives of `grant` and `read`. We first define `AC` as a specialization of the `SST` monad where we track file access permissions as a set of filenames both at the runtime level and – using implicits to relate the input and output states – at the type level.

```
-- | Token State Transformer ----------------------------------------
type TokM m1 m2 a = SST m1 m2 Tk a

-- | Start a stream, Get the next token unless the stream is done --
done  :: Tk
init :: [t:Tk]. TokM {} (store {} t ⊤) {v|v = t}
next :: [m:Map Tk Bool] → t:{Tk | (select m t) /\ (not (t = done))}
        → ([t':Tk]. TokM m (store (store m t ⊥) t' ⊤) t')

-- | Looping through all tokens -----------------------------------
client :: [m:Map Tk Bool] →
            t:{select m t} → TokM m {m'|select m' done} ()
client t = if t == done then pure ()
                        else do {t' ← next t; client t'}

-- | Starting the stream and running client ------------------------
main :: TokM {} {m|select m done} ()
main = do {t ← init; client t}
```

Figure 4 Verifying a Token Stream API.

An API for safe file access. We use the `AC` monad to develop an API that statically enforces compliance with an access control list (ACL). The `grant` primitive adds a file name to the access list, and `read` statically checks that the file is in the ACL and – for simplicity – just returns the string `"file␣contents"`. In conjunction with the implementations of `pure` and `»=` from Figure 2, the combinators can be used to verify `main` which, running with an initial empty access control list, grants permission to read a file then reads the file. If we accidentally tried to `read` from, say, `"secret-password.txt"`, the type checker would reject the program as unsafe.

Dynamic policies. Systems enforcing access control policies in practice [28] are not necessarily limited to a static policy – instead, they define a *checked read*, which checks at runtime whether a file is in the access control list, and then reads that file, returning a failure result if the permission check fails. We enable this via `canRead`, which determines if a file is in the dynamic ACL. This is reflected at the type level by passing in an implicit access control list `p` and specifying that `canRead` returns true iff its argument is in the ACL `p`. We then use `canRead` to define `safeRead`, which can be called with no conditions on the ACL. Instead, it uses `canRead` to dynamically check permissions, returning `Nothing` on failure. Crucially, to verify `main` and `safeRead` we do not need to tediously instantiate ghost variables, as Implicit Refinement Types automatically infer the suitable instantiations.

2.5 Token Stream

The Hoare State Transformer can be used to verify the token stream example of §1. In Figure 4 we show how to instantiate it with refinements over maps that track the validity of tokens. First, we define a specialization of the `SST` monad named `TokM` whose concrete state is a token (of type `Tk`) that tracks the last token we sent. `TokM`'s ghost state is a map of the status of *every* token. That is, `select m t` represents the proposition that the token `t` is *valid* (the next value to pass to the API): if `select m t = ⊤` (shorthand for `true`), then `t` is valid. Otherwise, `select m t = ⊥` (shorthand for `false`) means that the token `t` has been used and is now invalid.

```
-- | Singletons as resource counts ---------------------------------
data Tick t a = Tick a
type T t a = Tick {v:Int | v = t} a

-- | The Applicative Functor API ----------------------------------
<*>  :: [n:Int m:Int] → T n (a → b) → T m a → T (n + m) b
</>  :: [n:Int m:Int] → T n (a → b) → T m a → T (n + m + 1) b
pure :: x → T 0 a

-- | Appending two lists in a linear number of steps ---------------
 ++    :: xs:(List a) → ys:(List a)
                       → T (len xs) {v|len v = len xs + len ys}
[]       ++ ys = pure ys
(x:xs')  ++ ys = pure (x:) </> (xs' ++ ys)

-- | Mapping a costly function over a list -------------------------
mapA :: [n] → (a → T n b) → xs:List a → T (n * len xs) (List b)
mapA f []     = pure []
mapA f (x:xs) = pure (:) <*> f x <*> mapA f xs
```

Figure 5 Intrinsic Verification of Resource Usage.

An API for streaming tokens. The `TokM` monad lets us specify and verify the token streaming API from Section 1. First, we specify a special token `done` that represents the last token of the stream. On the other hand, `init` represents a *computation* that begins the stream using an implicit pair to capture that there is *some* valid token `t` resulting from the computation of `init`. Moreover, `init` starts with the empty map (with no stale tokens) to ensure that we may only begin the stream once.

The workhorse of this API is `next`. It first takes an implicit history parameter `m` representing all of the past tokens. We then check that the token `t` passed in is not the `done` token, and use `m` to constrain `t` to be valid (`select m t`). Finally, another implicit pair is used to produce the *prophecy* variable `t'` which is both the next value returned by the computation, and the next valid token as noted by the ghost state, which also marks the old token `t` invalid.

We can now develop the `client`, which recursively consumes the remaining stream of tokens. Were we to attempt to reuse the token `t` in the recursive call the program will be correctly rejected as unsafe. `main` kicks off the stream with `init` and consumes it using `client`. Thus, implicit refinements eliminate the tedium of manually instantiating ghosts.

2.6 Intrinsic Verification of Resource Usage

Next, we demonstrate how implicit refinement types can be used for specifying higher-order programs beyond the state monad: in particular, tracking resource usage. Figure 5 defines an applicative functor for counting resource usage in the same style as Handley et al. [19]. This API has both the standard application operator `<*>` and a resource-consuming application operator `</>`. The API counts the number of times we use the `</>` operator, which allows us to apply a function `f` with cost `n` to an argument `x` of cost `m`, incurring a total cost of `n + m + 1`.

Handley et al. [19] show these combinators can be used to verify properties about resource usage. We adapt their example of counting the recursive steps in `xs ++ ys`. At each recursive

step, we append another element `x` to the beginning of the list using `pure (x:)</>`. Ultimately `(++)` will use `len xs` applications of this operator to build this list, verifying that `(++)` is linear in the first argument.

Using this API we can further define the higher-order constant-resource combinator `mapA`, which allows us to map a function `f` of constant cost `n` over a list `xs`, and automatically verify that doing so costs `n * len xs`. This is easy to specify with implicit refinement types: the implicit argument lifts the output cost of the function argument so that we may relate it with the overall cost of calling `mapA`.

Two-State Specifications. It is worth pausing here to recall the standard technique for specifying effectful programs like the ones we have shown: the two-state specifications that allow expressing the input and output requirements of a particular computation. For instance, `fresh` (Figure 2) would be given a specification such as `requires (\s → ⊤) ensures (\s o s' → s' = s + 1 ∧ o = s)` where `s` and `s'` represent the input and output state and `o` represents the output of the computation. Notably, even if our language included such two-state specifications, specifying `mapA` would still require an extra parameter, as the relationship between the start and end "states" of `mapA` depends on the relationship between the start and end "states" of the `(a → T n b)` argument.

On the other hand, implicit refinements scale from capturing relations between the inputs and outputs of a single computation to relating a higher-order computation to its function argument(s) without having to "hardwire" some notion of two-states or pre/post conditions. Instead, they allow us to *name* the input and output worlds and *lift them to the top level*, which allows assertions (refinements) that span those states/worlds, including in examples such as `mapA`. The key contribution of Implicit Refinement Types then is that they work both for classic two-state specifications *and other* use-cases where two-state specifications prove cumbersome and allow the necessary extra parameter of functions like `mapA` to be instantiated automatically.

3 Programs

We start with a declarative static semantics for our elaborated core language λ^R. Our discussion here omits polymorphism as it is orthogonal to adding implicit types. (The full system can be found in the extended edition [40]).

3.1 Syntax

Figure 6 presents the syntax of our source language – a lambda calculus with refinement types, extended with implicit function and dependent pair types.

Types. of λ^R begin with base types `Int` and `Bool`, which are *refined* with a (boolean-valued) expression r to form refined base types $\{x : b \mid r\}$. Next, λ^R has dependent function types $x : t_1 \to t_2$. Dependent function types are complemented by *implicit* dependent function types $[x : t_1] \to t_2$, which are similar, except that the parameter x is passed *implicitly*, and does not occur at runtime. Dually, we have *implicit* dependent pairs $[x : t_1].t_2$, which represent a pair of values: The first, named x, of type t_1, is implicit (automatically determined) and does not occur at runtime. Meanwhile, the second is of type t_2 which may refer to x. We use τ to denote *unrefined* types.

Types

$b \quad ::= \quad \texttt{Int} \mid \texttt{Bool} \mid \cdots$

$t \quad ::= \quad \{x : b \mid r\} \mid x : t \to t \mid [x : t] \to t \mid [x : t].t$

Terms

$c \quad ::= \quad \mathit{false}, \mathit{true} \mid 0, 1, \ldots \mid \wedge, \vee, +, -, =, \leq, \ldots$

$e \quad ::= \quad c \mid x \mid \lambda x : t.e \mid e\ e \mid \texttt{let}\ x : t = e\ \texttt{in}\ e \mid \lambda^i x : t.e \mid \texttt{unpack}\ (x, y) = e\ \texttt{in}\ e$

Contexts

$\Gamma \quad ::= \quad \bullet \mid \Gamma, x : t \mid \Gamma, [x : t]$

Type Checking $\boxed{\Gamma \vdash e : t}$

$$\textsc{T-AbsI}\ \frac{\Gamma, [x : t_x] \vdash e : t}{\Gamma \vdash \lambda^i x : t_x.e : ([x : t_x] \to t)} \qquad \textsc{T-App}\ \frac{\Gamma \vdash e_1 : t \quad \Gamma \mid t \vdash e_2 : t'}{\Gamma \vdash e_1\ e_2 : t'} \qquad \textsc{T-Var}\ \frac{x : t \in \Gamma}{\Gamma \vdash x : t}$$

$$\textsc{T-Let-}\tau\ \frac{\Gamma \vdash e_x : t_x \quad \Gamma, x : t_x \vdash e : t \quad \Gamma \vdash t \quad \Gamma \vdash t_x \quad \lfloor t_x \rfloor = \tau}{\Gamma \vdash \texttt{let}\ x : \tau = e_x\ \texttt{in}\ e : t} \qquad \textsc{T-Unpack}\ \frac{\Gamma \vdash e_1 : [x' : t_1].t_2 \quad \Gamma, [x : t_1], y : t_2[x/x'] \vdash e_2 : t \quad \Gamma \vdash t}{\Gamma \vdash \texttt{unpack}\ (x, y) = e_1\ \texttt{in}\ e_2 : t}$$

Application Checking $\boxed{\Gamma \mid t_1 \vdash e : t_2}$

$$\textsc{AppI}\ \frac{\Gamma \mid t[e'/x] \vdash e : t' \quad \langle \Gamma \rangle \vdash e' : t_x}{\Gamma \mid [x : t_x] \to t \vdash e : t'} \qquad \textsc{App}^{\ e}\ \frac{\Gamma \vdash e : t_e \quad \Gamma \vdash t_e \preceq t_x \quad \Gamma, y : t_e \vdash t[y/x] \preceq t' \quad \Gamma \vdash t' \quad y\ \text{fresh}}{\Gamma \mid x : t_x \to t \vdash e : t'}$$

Subtyping $\boxed{\Gamma \vdash t_1 \preceq t_2}$

$$\preceq\textsc{-Base}\ \frac{[\![\Gamma]\!](\forall v_1 : b.r_1 \Rightarrow r_2[v_1/v_2])\ \text{is valid}}{\Gamma \vdash \{v_1 : b \mid r_1\} \preceq \{v_2 : b \mid r_2\}} \qquad \preceq^R_{\text{IPAIR}}\ \frac{\Gamma \vdash t_1 \preceq t_2'[e/x] \quad \langle \Gamma \rangle \vdash e : t_2}{\Gamma \vdash t_1 \preceq [x : t_2].t_2'}$$

Figure 6 Syntax and static semantics of λ^R.

Terms. of λ^R comprise *constants* (booleans, integers and primitive operations) and *expressions*. Let binders are half-annotated with either a refinement type to be checked, or a base type on which refinements are to be inferred.

In addition to explicit function abstraction, λ^R has the implicit λ-former $\lambda^i x : t.e$, where the parameter x represents a *ghost* value that can only appear in refinement types. Implicit functions are instantiated automatically, so there is no syntax for eliminating them. Similarly, implicit dependent pairs are introduced automatically, and thus have no introduction form in λ^R. Instead, implicit dependent pairs have an *elimination* form $\texttt{unpack}\ (x, y) = e_1\ \texttt{in}\ e_2$. Here, if e_1 is of type $[x : t_x].t$, then x is bound at type t_x and y is bound at type t in e_2. Just like with implicit functions the x represents a *ghost* value that may only appear in refinement types.

Though both implicit lambda and unpack terms are present in our model, in practice we handle their insertion by elaboration: we discuss this aspect of our implementation in §7. In light of this, we develop the theory of Implicit Refinement Types in terms of the fully elaborated expressions of the λ^R syntax.

Contexts. of λ^R, written Γ, comprise the usual ordered sequences of "corporeal" binders $x : t$, where x is visible in both terms and refinements, as well as *ghost binders* $[x : t]$, where x is only visible in refinements.

3.2 Static Semantics

Figure 6 provides an excerpt of the declarative typing rules for λ^R (the complete rules are in the extended edition [40]). Most of the rules are standard for refinement types [34] so we focus our attention on the novel rules regarding implicit types.

Type Checking. judgments of the form $\Gamma \vdash e : t$ mean "in context Γ, the term e has type t." The ghost binders in Γ, written $[x : t]$, reflect the ghostly, refinement-only nature of implicits. This distinction is witnessed by the rule [T-Var] which types term-level variables using only the corporeal binders $x : t$ in Γ. This ensures that implicit variables are erasable: they can only appear in types (specifications) and thus *cannot* affect computation.

With the separation of ghostly (erasable) implicit binders from corporeal (computationally relevant) binders, both the introduction rule for implicit functions [T-AbsI] and the elimination rule for implicit pairs [T-Unpack] are standard up to the ghostliness of binders. Implicit pairs are eliminated through "unpacking" as is typical for dependent pairs and existential types. The rule [T-Let-τ] demonstrates how we handle annotations at unrefined base types τ: we pick some well-formed refinement type t_x that erases to the base type $\lfloor t_x \rfloor = \tau$, and then use t_x as the bound type of x.

Lastly, we split off type checking of applications into an application checking judgment, in order to handle instantiations of implicit functions. In the rule [T-App] we use this additional judgment to check that the argument e_2 is compatible with the input type of e_1.

Application Checking. judgments of the form $\Gamma \mid t \vdash e : t'$ mean "when e is the argument to a function of type t the result has type t'". The (corporeal) application rule [App ^{e}], finds the type t_e of e, checks that this is consistent with the input type t_x of the function, and then creates a new name y to refer to e so that we may substitute it into the return type. However, y is not bound in Γ so we guess a type t', well-formed under Γ, for the entire term, and check that the return type $t[y/x]$ is a subtype of t'. The rule [AppI] checks implicit applications by *guessing* an expression e' at which to instantiate the implicit parameter. This e is only used at the refinement-level and is thus allowed to range over both corporeal and ghost binders in Γ, as described by the antecedent $\langle\Gamma\rangle \vdash e : t$. ($\langle\Gamma\rangle$ replaces each ghost binder $[x : t]$ in Γ with a corresponding corporeal binder $\langle[x : t]\rangle = x : t$.) The rule then continues along the spine of the application, further instantiating implicit parameters as necessary until we can apply the rule [App ^{e}].

Subtyping Judgments. of the form $\Gamma \vdash t_1 \preceq t_2$ mean "in context Γ, the values of t_1 are a subset of the values of t_2." Most of the rules are standard, with the base case [$\preceq$-Base] reducing to a *verification condition* (VC) that checks if one refinement *implies* another. [$\preceq^R_{\text{IPAIR}}$] serves as the "introduction form" for implicit pairs, and states that a term of type t_1 can be used as an implicit pair type if there is some expression e of type t_2 such that t_1 is a subtype of t_2' instantiated with that e. Note that this is similar to explicitly constructing the dependent pair with e as the first element.

$$
\begin{array}{rrcl}
\textit{Predicates} & r & ::= & \ldots \text{varies} \ldots \\
\textit{Types} & \tau & ::= & \ldots \text{varies} \ldots \\
\textit{Propositions} & p & ::= & \kappa(\overline{x}) \mid r \\
\textit{Existential Horn Clauses} & c & ::= & \exists x : \tau.c \mid c \wedge c \mid \forall x : \tau.p \Rightarrow c \mid p \\
\textit{First Order Assignments} & \Psi & ::= & \bullet \mid \Psi, r/x \\
\textit{Second Order Assignments} & \Delta & ::= & \bullet \mid \Delta, \lambda\overline{x}.r/\kappa
\end{array}
$$

Figure 7 Syntax of L^S.

4 Logic

We define the syntax and semantics of *verification conditions* (VCs) generated by rule [$\preceq$-Base]. Figure 7 summarizes the syntax of VCs, *Existential Horn Clauses* (EHC), which extends the NNF Horn Clauses used in Cosman and Jhala [10] with *existential* binders. *Predicates* r range over a decidable background theory. *Propositions* p include predicates and second order *predicate variables* $\kappa(\overline{x})$. *Clauses* c comprise quantifiers, conjunction, and propositions. In the syntax tree of clauses, there are two places a κ variable may appear: as a leaf (*head* position), or in an antecedent under a universal quantifier $\forall x : \tau.\kappa(\overline{y}) \Rightarrow c$, (*guard* position).

Dependencies of an EHC constraint c are the set E of pairs (κ, κ') such that κ appears in guard position in c and κ' appears in head position under that guard. When $(\kappa, \kappa') \in E$, we say that κ *depends on* κ', or, dually, κ' appears *under* κ. The *cycles* in a constraint c are nonempty sets S of predicate variables such that: for all $\kappa \in S$, there exists $\kappa' \in S$ such that $(\kappa, \kappa') \in E$. A set of predicate variables S is said to be *acyclic* in c, if all cycles in c contain at least one predicate variable not in S. A predicate variable κ is said to be *acyclic* in c, if no cycles in c contain κ. A constraint c is said to be *acyclic* if there are no cycles in c.

Semantics. [$\preceq$-Base] checks the *validity* of the formula obtained by interpreting contexts and terms in our constraint logic, (formalized by $[\![\cdot]\!]$ in the extended edition [40]). Contexts Γ yield a sequence of universal quantifiers for all variables bound at (interpretable) basic types. Recall that VCs do not contain predicate variables $\kappa(\overline{x})$. The restricted grammar of the VCs is designed to be amenable to SMT solvers, represented by an oracle $\mathsf{Valid}(c)$ that checks validity, defined at the end of this section.

We eliminate *predicate variables* κ via substitution, $c[\Delta]$ (defined in the extended edition [40]) that map them onto meta-level lambdas $\lambda\overline{x}.p$. We represent solutions to *existential binders* with a substitution (Ψ) binding existential variables to predicates. We define an *existential substitution* $c\{r/x\}$ recursively over c which removes the corresponding existential binder, using standard substitution to replace x with its solution r: $(\exists x : \tau.c)\{r/x\} \doteq c[r/x]$.

EHC Validity is then defined by the judgment $\Delta; \Psi \vDash c$. Intuitively, c is valid under the substitutions Δ, Ψ if the result of applying the substitutions yields a VC that is valid. We say that an EHC is *satisfiable*, written $\vDash c$ if *there exist* Δ and Γ such that $\Delta; \Gamma \vDash c$. We say that c and c' are *equisatisfiable* when $\vDash c$ iff $\vDash c'$.

5 Type Inference

The declarative semantics described in Figure 6 are decidedly non-deterministic. This is most evident in the rules for implicits, such as [AppI] and $[\preceq^R_{\text{IPAIR}}]$, where, out of thin air, we generate an expression to instantiate the implicit parameter. Additional non-determinism

Subtyping $\boxed{\Gamma \vdash t_1 <: t_2 \dashv c}$

$$\frac{c = \forall x : \tau. [\![r]\!] \Rightarrow [\![r'[x/y]]\!]}{\Gamma \vdash \{x : \tau \mid r\} <: \{y : \tau \mid r'\} \dashv c} \qquad \frac{\Gamma, z : t_2 \vdash t_1 <: t_2'[z/x] \dashv c \;\; z \text{ fresh}}{\Gamma \vdash t_1 <: [x : t_2].t_2' \dashv \exists z :: t_2.c}$$

Checking $\boxed{\Gamma \vdash e \Leftarrow t \dashv c}$

$$\text{C-Sub}\;\frac{\Gamma \vdash e \Rightarrow t' \dashv c \qquad \Gamma \vdash t' <: t \dashv c'}{\Gamma \vdash e \Leftarrow t \dashv c \wedge c'} \qquad \text{C-Let-}\tau\;\frac{\Gamma \vdash e_1 \Leftarrow \hat{t} \dashv c_1 \qquad \Gamma, x : \hat{t} \vdash e_2 \Leftarrow t \dashv c_2 \;\; \hat{t} = \mathsf{fresh}(\Gamma, \tau)}{\Gamma \vdash \texttt{let}\; x : \tau = e_1 \;\texttt{in}\; e_2 \Leftarrow t \dashv c_1 \wedge (x :: \hat{t} \Rightarrow c_2)}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow [x' : t_1].t_2 \dashv c_1 \quad \Gamma, [x : t_1], y : t_2' \vdash e_2 \Leftarrow t \dashv c_2 \qquad t_2' = t_2[x/x'] \quad c = c_1 \wedge (x :: t_1 \Rightarrow (y :: t_2' \Rightarrow c_2))}{\Gamma \vdash \texttt{unpack}\; (x, y) = e_1 \;\texttt{in}\; e_2 \Leftarrow t \dashv c} \qquad \frac{\Gamma \vdash e_1 \Rightarrow t' \dashv c_1 \qquad \Gamma \mid t' \vdash e_2 \ll t \dashv c_2}{\Gamma \vdash e_1\; e_2 \Leftarrow t \dashv c_1 \wedge c_2}$$

Synthesis $\boxed{\Gamma \vdash e \Rightarrow t \dashv c}$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x \Rightarrow t \dashv \top} \qquad \frac{\Gamma \vdash e_1 \Rightarrow t_1 \dashv c_1 \qquad \Gamma \mid t_1 \vdash e_2 \gg t_2 \dashv c_2}{\Gamma \vdash e_1\; e_2 \Rightarrow t_2 \dashv c_1 \wedge c_2}$$

Application Checking $\boxed{\Gamma \mid t \vdash e \ll t' \dashv c}$

$$\text{C-App-}\rightarrow\;\frac{\Gamma \vdash y \Leftarrow t_x \dashv c_1 \qquad \Gamma \vdash t[y/x] <: t' \dashv c_2}{\Gamma \mid x : t_x \rightarrow t \vdash y \ll t' \dashv c_1 \wedge c_2} \qquad \text{C-App-IFun}\;\frac{\Gamma, [z : t_x] \mid t[z/x] \vdash y \ll t' \dashv c \qquad z \text{ fresh}}{\Gamma \mid [x : t_x] \rightarrow t \vdash e \ll t' \dashv \exists z :: t_x.c}$$

Figure 8 Constraint Generation.

appears in rules like [T-Unpack], where a refinement type must be picked such that it is well-formed under the outer context Γ. This is required as the body of the unpack expression is checked under Γ extended with the binders x and y, but the type must be well-formed under Γ itself to ensure that these variables do not escape (since they may appear in the type t).

We account for all of the non-determinism of the declarative semantics with an algorithmic, bidirectional type inference system [32, 15], excerpts of which are shown in Figure 8 (the full rules are in the extended edition [40]). We split the declarative type checking judgments into two forms: synthesis ($\Gamma \vdash e \Rightarrow t \dashv c$) and checking ($\Gamma \vdash e \Leftarrow t \dashv c$) along with corresponding application synthesis ($\Gamma \mid t \vdash y \gg t' \dashv c$) and application checking ($\Gamma \mid t \vdash y \ll t' \dashv c$) forms. Synthesis forms produce the type as an output while checking forms take the type as an input. We also introduce an algorithmic subtyping judgment ($\Gamma \vdash t_1 <: t_2 \dashv c$). In addition to their other outputs, these judgments produce an EHC c as an output. The core of our inference algorithm is precisely in extending the restricted grammar of verification conditions to an EHC that captures the constraints on the non-deterministic choices. Inference then reduces to the satisfiability of the constraint c (as checked in §6).

5.1 Constraining Unknown Refinements

Consider the following λ^R program from §2.1:

$$\text{EXAMPLE} = \texttt{let}\; y : (\texttt{Bool} \rightarrow \texttt{Int}) = \lambda z : \texttt{Bool}.1\;\texttt{in}\;\; \text{foo}\; y.$$

recalling that foo has the type $[n : \texttt{Int}] \rightarrow (\texttt{Bool} \rightarrow \{v : \texttt{Int} \mid v = n\}) \rightarrow \texttt{Unit}$. We wish to check that this program is safe by checking that it types with type `Unit`.

To type EXAMPLE in our declarative semantics, we first need to apply the rule [T-LET-τ] which non-deterministically chooses a t_x, well-formed under Γ ($\Gamma \vdash t_x$), such that t_x is consistent with the base type ($\lfloor t_x \rfloor = \texttt{Bool} \to \texttt{Int}$). To capture picking this refinement type we employ predicate variables that represent unknown refinements, as is standard in the refinement type inference literature [23, 34]. As we know the type we wish to give EXAMPLE, we will focus on the checking rule [C-LET-τ]. We generate a fresh refinement type using fresh, which takes as input a type t and the current context Γ and then produces a refinement type, where each base type is refined by a *fresh* predicate variable $\kappa(\overline{x})$, where $\overline{x}$ are all of the variables bound in the context, all of which can appear in refinements at this location. In our example, this would give the type $\hat{t} = z : \{v : \texttt{Bool} \mid \kappa_1(v)\} \to \{v : \texttt{Int} \mid \kappa_2(z, v)\}$.

We then check $\lambda z : \texttt{Bool}.1$ at the refinement type $\hat{t}$. Now, we use the rule [C-SUB] to synthesize a type for this term and then use subtyping to check that the synthesized type is subsumed by $\hat{t}$. The synthesized type will be $z : \{v : \texttt{Bool} \mid \kappa_3(v)\} \to \{v : \texttt{Int} \mid v = 1\}$ and the subtyping check will generate the following constraint which is equivalent to (1) in Figure 1a modulo the administrative predicate variable κ_3 and a simplification of the unconstrained κ_1:

$$\wedge\, \forall v.\ \kappa_1(v) \Rightarrow \kappa_3(v)$$
$$\wedge\, \forall z.\ \kappa_1(z) \Rightarrow \forall v.\ v = 1 \Rightarrow \kappa_2(z, v)$$

There is a subtlety in how [C-LET-τ] generates the quantified subformula $\forall z.\ \kappa_1(z) \Rightarrow \cdots$: this formula is generated by the clause $x :: \hat{t} \Rightarrow c_2$ where the double colon represents a *generalized implication* that drops any variables quantified at non-base types (as only base types are interpreted into the refinement logic).

$$x :: \{x : b \mid r\} \Rightarrow c \doteq \forall x : b.r \Rightarrow c \qquad x :: t \Rightarrow c \doteq c$$

5.2 Constraining Implicit Application

The predicate variables let us capture guessed refinement types as second order constraints. Next we turn to checking the *implicit* application foo y. foo has the type $[n : \texttt{Int}] \to (\texttt{Bool} \to \{v : \texttt{Int} \mid v = n\}) \to \texttt{Unit}$, so the declarative semantics arbitrarily picks an expression e of type `Int` to instantiate n. In the algorithmic type system, we capture the constraints on this choice with the existential quantifiers of our EHC. This is shown in the application checking rule [C-APP-IFUN] (the corresponding application synthesis rule [S-APP-IFUN] appears in the extended edition [40]).

Recall that the judgment $\Gamma \mid t \vdash e \ll t' \dashv c$ says that we are checking an application of a term of type t to an argument e and require that the application has the type t'. Here, we are checking foo y against the type `Unit`. Our bidirectional rule [C-APP-IFUN] "guesses" the instantiation by generating a fresh variable n and binding it at the type `Int`. This variable is added to the context as predicate variables may depend on it. We then generate a constraint $\exists n :: \texttt{Int}.c$ which says that our guessed n must be consistent with c, the constraint generated by continuing to check down the abstract syntax tree (along the spine of the application of the revealed function type $t[y/x] = (\texttt{Int} \to \{v : \texttt{Int} \mid v = n\}) \to \texttt{Unit}$). $\exists n :: \texttt{Int}.c$ is the existential counterpart to the generalized implication $n :: \texttt{Int} \Rightarrow c$:

$$\exists x :: \{x : b \mid r\}.c \doteq \exists x : b.(r \wedge c) \qquad \exists x :: t.c \doteq c$$

This is now a concrete function type and the rule [C-APP-$\to$] will check that the argument y has the type $\texttt{Int} \to \{v : \texttt{Int} \mid v = n\}$ and that the type `Unit` is a subtype of `Unit`. Checking the implicit and then concrete application thus generates the constraints $\exists n.(\forall v.\ \top \Rightarrow \kappa_1(v) \wedge \forall v.\ \kappa_2(\_, v) \Rightarrow v = n)$.

Combining these constraints with those generated from checking $\lambda z : \texttt{Bool}.1$, we get constraints equivalent to those in Figure 1a. A satisfying assignment is:

$$\Delta = [\lambda z, v.v = 1/\kappa_1, \lambda v.\top/\kappa_2, \lambda v.\top/\kappa_3] \qquad \Psi = [1/v]$$

Satisfying solutions to the predicate variables and existential constraints give instantiations to the angelic choices of refinement types and implicit arguments respectively. This gives us the following soundness theorem for our type inference algorithm (where the function $\mathsf{kvars}(c)$ returns the set of predicate variables in c):

▶ **Theorem 1** (Soundness of Type Inference). *If* $\bullet \vdash e \Rightarrow t \dashv c$, $\Delta; \Psi \vDash c$, *and* $\mathsf{kvars}(t) \subseteq \mathsf{domain}(\Delta)$, *then* $\bullet \vdash e : \Delta(t)$.

6 Solving

The constraints generated by algorithmic type checking have both predicate variables and alternating universal and existential quantifiers. We must provide solutions to both predicate variables and existential variables before we can use SMT solvers to check the validity of a VC (§4). We compute solutions in four steps:

1. We transform the EHC by skolemization to replace existential variables with universally quantified predicate variables and inhabitation side conditions.
2. We eliminate the original predicate variables.
3. We solve the skolem predicate variables.
4. Finally, we check the inhabitation side conditions.

Weakening and Strengthening. A function f on constraints is a *strengthening* when $\forall c.\ f(c) \Rightarrow c$. A function f on constraints is a *weakening* when $\forall c.\ c \Rightarrow f(c)$. We prove that, if the transformations in steps 1 and 2 above are both weakening and strengthening, our algorithm produces a verification condition that is *equisatisfiable* with the original constraint, *i.e.* our algorithm is sound and complete.

Separable Constraints. An EHC c is *separable* if it can be written as a conjunction $c_1 \wedge c_2$, where c_1 is an NNF Horn clause and c_2 is an acyclic EHC. The following theorem exactly characterizes separable EHCs:

▶ **Theorem 2.** *c is separable iff there are no cyclic κs under existential binders.*

There are standard partial techniques for solving cyclic NNF Horn clauses [6, 10] so the task of solving a separable EHC can be split into applying one of these existing techniques and then solving the acyclic EHC. Consequently, all a programmer must do to make constraints separable is provide either a local solution to an implicit variable via an explicit value or a solution to a cyclic predicate variable (*e.g.* by providing a type signature for a recursive function.)

Thus, in the sequel, we focus on the remaining problem: solving an acyclic EHC. We use the acyclic EHC from Figure 1a (reproduced below) as a running example. Recall that this is a simplified version of the constraints generated during type inference on the program EXAMPLE in §5.1 and §5.2.

$$\wedge\ \forall z.\ \top \Rightarrow \forall \nu.\ \nu = 1 \Rightarrow \kappa(\nu) \tag{3}$$

$$\wedge\ \exists n. \forall \nu.\ \kappa(\nu) \Rightarrow \nu = n \tag{4}$$

Step 1: Skolemization. We use the function skolem to transform the EHC c to a conjunction of an NNF Horn Clause $\mathsf{noside}(\mathsf{skolem}(\emptyset, c))$ and side conditions $\mathsf{side}(\mathsf{skolem}(\emptyset, c))$. This differs from textbook skolemization in two important ways: We replace each existential quantifier $\exists n.c$ with Skolem predicates $\forall n.\pi_n(n, \overline{x}) \Rightarrow c$ rather than Skolem functions, so that we can synthesize a (declarative) relation rather than a function. As a result, we must still check to make sure that this relation is inhabited. We do so by producing the side condition $\exists n.\pi_n(n, \overline{x})$. Our transformation Skolemizes the existential binding (4) of our example as follows:

$$\wedge\ \forall z.\ \top \Rightarrow \forall \nu.\ \nu = 1 \Rightarrow \kappa(\nu) \tag{5}$$
$$\wedge\ \forall n.\pi(n) \Rightarrow \forall \nu.\ \kappa(\nu) \Rightarrow \nu = n \tag{6}$$
$$\wedge\ \exists n.\pi(n) \tag{7}$$

This transformation will be crucial later: giving a name to π allows us to separate the inhabitation and sufficiency constraints on n.

skolem yields an NNF that has two classes of predicate variables: Skolem predicates corresponding to existential binders (written π_n) that have an inhabitation side condition, and predicate variables corresponding to unknown refinements (written κ). The π_n only appear negatively, so the standard technique of finding the least fixed point solution [34, 10] would simply return $\bot$, which will fail the inhabitation side conditions. Instead, we would like to compute the *greatest fixed point* solution for each π_n, but, for efficiency reasons, do not wish to compute the greatest fixed point solution of every predicate variable. Fortunately Cosman and Jhala [10] show that acyclic predicate variables can be eliminated one by one. We explain first how to eliminate κ variables and then how to eliminate π variables.

Step 2: Eliminating κ-Variables from c. Procedure elim1 of [10] eliminates each individual acyclic predicate variable, κ, in an NNF Horn Clause. Briefly, given a predicate variable κ in an NNF Horn Clause c, the procedure computes the strongest solution for κ: $\mathsf{sol}\kappa(\kappa, c)$, and then substitutes the solution into c. For the single κ in our example this solution $\mathsf{sol}\kappa(\kappa, c)$ is $\lambda x.(\exists z'.\top \wedge (\exists v'.v' = 1 \wedge v' = x))$. After substitution and simplification we get

$$\wedge\ \forall n.\pi(n) \Rightarrow \forall v.\top \Rightarrow \forall z'.\top \Rightarrow \forall v'.v' = 1 \Rightarrow v = v' \wedge v = n$$
$$\wedge\ \exists n.\pi(n)$$

Step 3: Eliminating Skolem Variables. elim1 removes all the κ predicate variables leaving only the π_n variables inserted by skolem. The inhabitation side conditions require we find the greatest fixed point (GFP) solution to these variables to ensure we do not spuriously eliminate witnesses. As π_n only appears negatively (in guards), the GFP is the conjunction of every c appearing as $\forall n.\pi_n(n, \overline{x}) \Rightarrow c$. For a given π_n appearing in the constraint c', we write this GFP as $\mathsf{def}\pi(\pi_n, c')$, the *defining constraint* of π_n.

A first challenge arises in that we wish to use these solutions to *eliminate* the Skolem predicate variables, but $\mathsf{def}\pi(\pi_n, c')$ is a conjunction of clauses featuring quantifiers. As the Skolem variables appear in guards, we must transform these c into an equisatisfiable predicate p before we may substitute, so we parameterize our elimination algorithms with a quantifier elimination algorithm qe that handles this task. qe can vary with the particular domain and, for domains that do not admit quantifier elimination or where it is infeasible, we instead use an approximation described in §6.1.

$\mathsf{elim}\pi^*_{qe}$	:	$P^C \to \overline{\Pi} \times C^{\Pi} \times C \to C$
$\mathsf{elim}\pi^*_{qe}([], \sigma, c)$	$\doteq$	c
$\mathsf{elim}\pi^*_{qe}(\pi_n : \overline{\pi}, \sigma, c)$	$\doteq$	$\mathsf{elim}\pi^*_{qe}(\overline{\pi}, \sigma, c[\lambda\overline{x}.p/\pi_n])$
where p	$=$	$qe(\mathsf{sol}\pi_{qe}(\{\pi_n\}, \sigma, \sigma(\pi_n)))$
$\mathsf{sol}\pi_{qe}$	:	$P^C \to \overline{\Pi} \times C^{\Pi} \times C \to C$
$\mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, \forall n.\pi_n(n, \overline{x}) \Rightarrow c)$		
$\mid \pi_n \in \overline{\pi}$	$\doteq$	$\mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, c)$
$\mid \pi_n \notin \overline{\pi}$	$\doteq$	$\forall n.p \Rightarrow \mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, c)$
where p	$=$	$qe(\mathsf{sol}\pi_{qe}(\overline{\pi} \cup \{\pi_n\}, \sigma, \sigma(\pi_n)))$
$\mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, \forall x.p \Rightarrow c)$	$\doteq$	$\forall x.p \Rightarrow \mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, c)$
$\mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, c_1 \wedge c_2)$	$\doteq$	$\mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, c_1) \wedge \mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, c_2)$
$\mathsf{sol}\pi_{qe}(\overline{\pi}, \sigma, p)$	$\doteq$	p

Figure 9 Eliminating π Variables and their Side Conditions.

A second technical challenge arises en route to our solving algorithm: $\mathsf{def}\pi(\pi_n, c')$ may contain *other* π variables and may contain cycles involving π variables. Fortunately, recursive Skolem variables are redundant:

▶ **Lemma 3.** *If π_n is a predicate variable inserted by* skolem*, then $\vDash \forall n.\pi_n(n, \overline{x}) \Rightarrow c$ iff $\vDash \forall n.\pi(n, \overline{x}) \Rightarrow c[\lambda\_.\top/\pi]$.*

This lets us to break cycles by ignoring recursive occurrences of Skolem variables.

With this result in hand, we develop the procedure $\mathsf{sol}\pi_{qe}$, as shown in Figure 9. $\mathsf{sol}\pi_{qe}$ recursively eliminates Skolem variables from the defining constraint of π_n, using qe to transform a nested Skolem variable's defining constraint (tracked in the map σ) into a substitutable predicate. The first argument $\overline{\pi}$ tracks the seen Skolem variables, treating them as if they were solved to $\top$ when they next occur.

$\mathsf{sol}\pi_{qe}$ is used in the procedure $\mathsf{elim}\pi^*_{qe}$ which eliminates all Skolem variables from the constraint one π_n at a time. $\mathsf{elim}\pi^*_{qe}$ simply calls $\mathsf{sol}\pi_{qe}$ on the defining constraint of π_n and then qe's the result (now free of Skolem variables) before substituting the returned predicate as the solution for π_n.

Regardless of the properties of qe we have the following soundness theorem:

▶ **Theorem 4.** *If $c' = \mathsf{skolem}(\emptyset, c)$, $\overline{\pi}$ is the set of all Skolem variables in c', c' has no other predicate variables, $\sigma(\pi_n) = \mathsf{def}\pi(\pi_n, c')$ for all $\pi_n \in \overline{\pi}$, and $\vDash \mathsf{elim}\pi^*_{qe}(\overline{\pi}, \sigma, c')$ then $\vDash c$.*

If qe is a strengthening and a weakening, the converse also holds and our algorithm is complete.

Step 4: Putting It All Together. The procedure safe (defined in the extended edition [40]) puts together all the pieces to automatically check the validity of acyclic EHC constraints c. Like $\mathsf{sol}\pi$ and $\mathsf{elim}\pi^*$, safe is parameterized by the quantifier elimination procedure qe. safe first Skolemizes the constraint c and then solves the original predicate variables by repeated applications of elim1. Next, safe collects the defining constraints of the Skolem predicate variables and uses $\mathsf{elim}\pi^*_{qe}$ to solve and eliminate the Skolem variables. This leaves us with a predicate-variable-free constraint, but still featuring nested existential quantifiers from

Table 1 Comparison of systems: ✓on "Spec" and "Check" respectively indicate that the specification can be written and that the code can be verified by automatically instantiating the implicit parameters.

Tool / Case Study	Mist				MoCHi [41]		F* Implicits [26]	
	LOC	Time (s)	Spec	Check	Spec	Check	Spec	Check
incr	8	0.01	✓	✓	✓	✗	✓	✗
sum	5	0.01	✓	✓	✓	✓	✓	✗
repeat	86	0.28	✓	✓	✓	✗	✗	✗
d_2 [41]	8	0.07	✓	✗	✓	✓	✓	✗
Resources								
incrState	28	0.65	✓	✓	✓	✓	✓	✓
accessControl	49	0.36	✓	✓	✓	✗	✓	✓[1]
tick [19]	85	0.03	✓	✓	✗	✗	✓	✓
linearDSL	20	0.03	✓	✓	✗	✗	✓	✓[1]
State Machines								
pagination	79	0.3	✓	✓	✗	✗	✗	✗
login [7]	121	0.09	✓	✓	✗	✗	✗	✗
twoPhase	135	0.86	✓	✓	✗	✗	✗	✗
tickTock	112	0.14	✓	✓	✗	✗	✗	✗
tcp	332	115.73	✓	✓	✗	✗	✗	✗

the inhabitation side conditions. Here, we use qe once again to eliminate the existential quantifiers, leaving us with a VC ripe to be passed to an automated theorem prover to verify validity. If this VC is valid, we deem c *safe*. If qe is a strengthening then safe is sound and if qe is additionally a weakening then safe is complete, proved in the extended edition [40].

▶ **Theorem 5.** *Let c be an acyclic constraint. If qe is a strengthening then $\mathsf{safe}(c)$ implies $\vDash c$. If qe is also a weakening then $\mathsf{safe}(c)$ iff $\vDash c$.*

6.1 A Theory-Agnostic Approximation to qe

We cannot, in general, provide a quantifier elimination that is a strengthening and a weakening (since we are agnostic to the set of theories) especially as some theories do not admit a decidable quantifier instantiation! However, since SMT theories work by equality propagation, we can make use of equalities between theory terms without making any additional assumptions about the theories themselves. Therefore, we include a theory-agnostic quantifier elimination strategy over equalities.

Given the defining constraint c of some $\pi_n(n, \overline{x})$, our strategy computes the (well-scoped) congruence closure of the variables n and $\overline{x}$ using the body of c. This set of equalities is then used as the solution to π_n. To eliminate the existential side condition $\exists n.\pi_n(n, \overline{x})$ we note that a sound approximation is to find a solution for n. We search for this solution within the set of equalities. If there is not one, we return $\bot$ and verification fails. In Section 7 we evaluate this incomplete quantifier elimination on a number of benchmarks and demonstrate its real-world effectiveness.

7 Evaluation

We implement our system of Implicit Refinement Types in a tool dubbed Mist, evaluate Mist using a set of illustrative examples and case studies (links to full examples elided

for DBR), and compare Mist against the existing state of the art, in order to answer the following questions:

Q1: Lightweight Verification Are implicits in conjunction with the theory-agnostic instantiation procedure sufficient to verify programs with IRTs, even without using heavyweight instantiation techniques such as domain-specific solvers and synthesis engines?

Q2: Expressivity Can we use implicits to encode specifications that would've otherwise required the use of additional language features?

Q3: Flexibility Do they allow *automated verification* in places where unification-based implicits and CEGAR-based extra parameters do not?

Implementation. Mist extends our language (§3) with polymorphism and type constructors, and omits concrete syntax for implicit lambdas and unpacks. Instead, implicit parameters appear solely in specifications – that is, refinement *types* – and do not require implementation changes to the code.

Inserting implicit lambdas is straightforward: when checking that a term has an implicit function type, insert a corresponding implicit lambda. Inserting implicit unpacks is more interesting: we need to unpack any term of implicit pair type before we use it. For instance, if e has the type $[x : t_1].t_2$, then we must unpack e to extract the corporeal (t_2) component of the pair before we can apply a function to it: $e_f\ e$ becomes $\texttt{unpack}\ (x, y) = e\ \texttt{in}\ e_f\ y$. This transformation ensures that the ghost parameter (x in the above term) is in context at the use site of the implicit pair. To automate this procedure, we use an Anf-like transformation that restricts A-normalization to terms of implicit pair type.

As is common in similar research tools, Mist handles datatypes by axiomatizing their constructors. A production implementation would treat surface datatype declarations as sugar over these axioms.

Polymorphism. Mist also includes support for limited refinement polymorphism. For simplicity, we left refinement polymorphism out of our formalism in §3.2 – it is orthogonal to the addition of implicit parameters and pairs – but we did include it in our implementation.

Refinement polymorphism alleviates some issues with phantom type parameters. First, when using phantom type parameters, core constructors cannot be directly verified and must be assumed to have the given type. Moreover, we can specify the semantics of our stateful APIs in terms of *e.g.* the `HST` type, but the meaning of the arguments to the HST type operator are determined only by its use. In contrast, refinement polymorphism brings the intended semantics of HST from the world of phantom parameters into the semantics of the language itself. The semantics of HST are reflected with the type:

```
type HST = rforall p q. forall a. p → (q, a)
```

where `rforall` ranges over refinement types and `forall` ranges over base types. The more sophisticated refinement polymorphism [42] present in existing refinement systems would use the type:

```
type HST p q s a = {w:s | p w} → ({w':s | q w'}, a)
```

Here, `p` and `q` are predicates on the state type `s`.

[1] Verification requires annotating a simple fact about sets as F* does not include a native theory of sets.

Refinement polymorphism further makes core constructors and API primitives themselves subject to verification: the `rforall` version of HST above allows the direct verification of `get` as

```
get : forall s. [w:s] → HST {v:s | v = w} {v:s | v = w} s
get = \s → (s, s)
```

Comparsion. We compare Mist to higher-order model checker MoCHi [41], and F*'s support for implicit parameters[2] [36, 26]. Both systems, like Mist and unlike foundational verifiers such as Idris [8] and Coq [39], are designed for lightweight, automatic verification. MoCHi aims to provide complete verification of higher-order programs by automatically inserting extra (implicit) parameters. Whereas Mist has users write refinement type specifications (with *explicit* reference to implicit types), MoCHi's specifications are implemented as assertions within the code. F*'s type system is a mix of a Martin-Löf style dependent type system with SMT-backed automatic verification of refinement type specifications. There is no formal specification of implicit parameters in F*. They are implemented by unification as part of Martin-Löf typechecking. This is in contrast to IRTs' integrated approach, that uses information from refinement subtyping constraints for instantiation.

Our comparison, summarized in Table 1, illustrates, via a series of case studies, the specifications that can be written in each system (the Spec column) and whether they can find the necessary implicit parameter instantiations (the Check column). As each tool is designed to be used for lightweight verification, we write the implementation and then separately write the specifications. We do not rewrite the implementations to better accommodate specifications, though for MoCHi we insert assertions within the code as necessary.

7.1 Q1: Lightweight Verification

We evaluate whether IRTs allow for modular specifications that would otherwise be inexpressible with plain refinement types. We do this via a series of higher-order programs that use implicits for lightweight verification. These programs are designed to capture core aspects of various APIs and how IRTs permit specifications that can be automatically verified in a representative client program using the API. Notably, for all of these examples Mist only uses the theory-agnostic instantiation procedure from §6.1 and does not employ heavyweight instantiation techniques such as domain-specific solvers or synthesis engines.

Higher-Order Loops. Repeat, defines a loop combinator `repeat` that takes an increasing stateful computation `body` and produces a stateful computation that loops `body count` times. Its signature in Mist is:

```
repeat :: body:([x:Int] → ([y:{v:Int | v > x}]. SST x y Int Int))
       → count:{v:Int | v > 0}
       → ([q:Int] → ([r:{v:Int | v > q}]. SST q r Int Int))
```

which says that the input and output are stateful computations whose history and prophecy variables together guarantee that the computation will leave the state larger than it started. In the first line the implicit function argument `x` is externally determined and captures the state of the world before the `body` computation, while the implicit pair component `y` captures that `body` updates the state of the world to some (unknown) larger value.

[2] Note that F*'s main verification mechanism is Dijkstra Monads [38], which we do not compare against in our evaluation. We discuss trade offs between Dijkstra Monads and Implicit Refinement Types in §8.

The implicit pairs are necessary to specify `repeat`, as the updated state is determined by an internal choice in `body`. REPEAT also demonstrates how implicit pairs can specify loop invariants (here, upward-closure) on higher-order stateful programs. Though `repeat` is specified using implicit pairs, MIST does *not* require that arguments passed to `repeat` be specified using implicit pairs. In REPEAT we define a function `incr` with the type `[x:Int] → SST x (x + 1) Int Int` that increments the state. MIST will appropriately type check and verify `repeat incr` as Implicit Refinement Types automatically account for the necessary subtyping constraints to ensure that `incr` meets the conditions of `repeat`.

State Machines. The next set of examples demonstrate that IRTs enable specification verification of state-machine based protocols. These are ubiquitous, spanning appications from networking protocols to device driver and operating system invariants [7]. IRTs let us encode state machines as transitions allowed from a ghost state, using the Hoare State Monad (2.3). LOGIN [7], which models logging into a remote server, served as our initial inspiration for studying this class of problems. We verify that a client respects the sequence of connect, login, and only then accesses information.

TICKTOCK verifies that two `ticker` and `tocker` processes obey the specification standard to the concurrency literature [35]. Here we show the implementation of `tocker`.

```
tocker = \c → do
msg ← recv c
if msg = tick then send c tock else assert False
```

Ghost parameters on `send` and `recv` track the state machine and ensure messages follow the tick-tock protocol. If `send c tock` is changed to `send c tick` the program is appropriately rejected.

TWOPHASE is a verified implementation of one side of a two-phase commit process. This example serves as a scale model for TCP, a verified implementation of a model of a TCP client performing a 3-way handshake, using the TCP state machine[33].

PAGINATION is an expanded version of our stream example from Section 2, and models the AWS S3 pagination API [1]. This example shows that our protocol state machine need not be finite, as it is specified with respect to an *unbounded* state machine.

Quantitative Resource Tracking. The next set of examples reflect various patterns of specifying and verifying *resources* and demonstrate that Implicit Refinement Types enable lightweight verification of these patterns. The examples show how MIST handles specification and automatic verification when dealing with resources such as the state of the heap and quantitative resource usage. In §2.4 we saw how Implicits enable specifying the access-control API from Figure 3; the ACCESSCONTROL example verifies clients of this API. In contrast, TICK (as shown in §2.6) follows Handley et al. [19] in defining an applicative functor that tracks quantitative resource usage. The key distinction from Handley et al. [19] is that the resource count exists only at the type level. This example generalizes: we can port any of the *intrinsic verification* examples from that work to use implicit parameters instead of explicitly passing around resource bounds.

LINEARDSL embeds a simple linearly typed DSL in MIST. It allows us to embed linear terms in MIST, with linear usage of variables statically checked by our refinement type system. The syntactic constructs of this DSL are smart constructors that take the typing environments as implicit parameters, enforcing the appropriate linear typing rules.

```
incr :: [n:Int] → (Int → SInt n) → SInt (n + 1)
incr f = (f 0) + 1

test1 :: SInt 11                    test2 :: m:Int → SInt (m + 1)
test1 = incr (\x → 10)              test2 m = incr (\x → m)
```

Figure 10 A higher order increment function.

Mist enables specification and automatic verification of this diverse range of examples, and in fact only requires the theory-agnostic instantiation procedure to find the correct implicit instantiations. This demonstrates that IRTs enable lightweight verification of a variety of higher-order programs, and that they are widely useful even without a domain specific solver or heavyweight synthesis algorithm.

7.2 Q2: Expressivity

We compare the expressivity of Implicit Refinement Types as implemented in Mist to the assertion-based verification of MoCHi and the implicit parameters of F*. First, merely the fact that we allow users to explicity write specifications using implicit parameters allows us to write specifications we otherwise could not. In particular, this is witnessed by the tick and linearDSL examples that specify resource tracking, a non-functional property. MoCHi cannot express the specifications for these because there is no combination of program variables that computes the (non-functional) *usage* properties used in these specifications.

Second, Implicit Pair Types add expressivity by allowing us to write specifications against choices made *internal* to functions that we wish to reason about. For example, repeat uses implicit pairs to specify a loop invariant. This example can't be done with F*'s implicits, as, absent implicit pairs, we cannot bind to the return value of the loop body. As a result, in F* we cannot verify this example with implicits alone: we would have to bring in additional features such as the Dijkstra monad [38].

Similarly, *none* of the protocol state machine examples can be encoded with implicit functions alone. Implicit pairs are required to write specifications against choices made by other actors on the protocol channel. As a result, these examples also cannot be encoded in F*'s implicits. These state machine specifications cannot even be expressed in MoCHi as they crucially require access to ghost state, which MoCHi does not support.

7.3 Q3: Flexibility

We compare the flexibility that our refinement-integrated approach gives us to solve for implicit parameters relative to that of other systems. We focus on the features of our semantics and our abstract solving algorithm from Section 6.1 independently of the choice of quantifier elimination procedure, which we examined above. We illustrate these differences with several examples. incr is the program from Figure 10. In MoCHi the specification is given by assertions that `test1` and `test2` are equal to `11` and `m + 1` respectively. incrState generalizes incr to track the integer in the singleton state monad instead of a closure. sum is similar to `incr` except that it takes two implicit arguments and two function arguments, returning the sum of the two returned values:

```
sum :: [n, m] → (Int → SInt n) → (Int → SInt m) → SInt (n + m)
sum f g = (f 0) + (g 0)

test :: SInt 11
test = sum (\x → 10) (\y → 1)
```

All three tools can specify the program. Mist and MoCHi successfully instantiate the implicit parameters needed for verification. D2 is an example from the MoCHi [41] benchmarks that loops a nondeterministic number of times and adds some constant each time. Unlike Mist, MoCHi is unable to solve accessControl, as CEGAR is notoriously brittle on properties over the theory of set-operations.

Comparsion to MoCHi. MoCHi fails to automatically verify incr, but it succeeds if either `test1` or `test2` appear individually. This is partly due to the unpredictability of MoCHi's CEGAR loop [21] (as it succeeds in verifying incrState), and partly due to the fact that MoCHi attempts to infer specifications (and implicit arguments) *globally*, which can lead to the anti-modular behavior seen here. In contrast, by introducing implicit arguments as a user-level specification technique, Mist permits modular verification: `test1` and `test2` generate two separate quantifier instantiation problems that Mist solves locally.

In D2, Mist *fails* to solve the constraints generated by implicit instantiation as they involve systems of inequalities, which our theory-agnostic quantifier elimination does not attempt. However, Mist captures the full set of constraints and could, with a theory specific solver like MoCHi's, verify D2. Concretely, D2 yields a constraint of the form $\exists x.true \Rightarrow x > 3$. Mist could discharge this obligation by instantiating the abstract algorithm of Section 6 with either the solver from MoCHi [41] or EHSF[5] instead of the theory-agnostic one above.

Comparsion to Unification. The results of the comparison to F*'s unification-based implicits are summarized in Table 1. Implicit Refinements are more flexibile as F* attempts to solve the implicits purely through unification, *i.e.* without accounting for refinements. In contrast, Mist generates subtyping (implication) constraints that are handled separately from unification. When the unification occurs under a type constructor, then unification can succeed. For example, F* verifies incrState because elevating the ghost state to a parameter of the singleton state type constructor makes it "visible" to F*'s unification algorithm. However, if there is *no* type constructor to guide the unification, F* must rely on higher-order unification, which is undecidable in general and difficult in practice. For this reason, in the incr example from Figure 10, F* fails to instantiate the implicit arguments needed to verify `test1` and `test2`, even if we aid it with precise type annotations on `\x → 10` and `\x → m`, and F* fails to verify the sum example for the same reason. By contrast, Mist can take advantage of the refinement information to solve for the implicit parameters. Finally, F*'s unification fundamentally cannot handle an example like MoCHi's D2, as unification will not be able to find an implicit instantiation when it is only constrained by inequalities.

8 Related Work

Verification of Higher-Order Programs. As discussed in Section 7, F* [37] is another SMT-aided higher-order verification language. Its main support for verifying stateful programs is baked in via Dijkstra Monads [38]. When it comes to verifying higher-order stateful programs, Dijkstra Monads enable F* to scale automatic verification up to complex invariants. However, like other two-state specification techniques, they are not on their own flexible enough to handle higher-order computations such as `mapA`, when higher-order computations are composed in richer ways than simple Kleisli composition. The key difference is that Implicit Refinement Types work for both classic two-state specifications *and other* use-cases where two-state specifications prove cumbersome – IRTs add value even to a system that already includes

two-state specifications by allowing the necessary extra parameter of functions like `mapA` to be instantiated automatically. Moreover, IRTs do so while maintaining the key properties of refinement type systems: (1) SMT-driven automatic verification and (2) refinement type specifications are added atop an existing program without requiring code changes. In this way IRTs also differ from systems that use "full-strength" two-state specifications with dependent types such as VST[3] or Bedrock[9].

We have started implementing implicit refinement types in LiquidHaskell, a very similar system to Mist that includes both manual [45] and automated [42] facilites for higher order verification.

F* has both implicit parameters and refinement types, but F*'s implicits have no formal description, and instantiation is independent of refinement information. On the other hand, we lack first-class invariants, but future work may be able to alleviate this by abstracting over refinements *a la* Vazou et al. [43].

Dafny [24] supports specifications with ghost variables, but the user must explicitly craft triggers to perform quantifier instantiation when the backing SMT solver's heuristics cannot, and must manually pass and update ghost variables.

Implicit Parameter Instantiation. Since finding an instantiation of implicit parameters is, in general, undecidable, different systems make different tradeoffs: While Haskell and Scala only perform type-directed lookup, Idris [8] resolves implicits via first-order unification, with a default value or by a fixed-depth enumerative program synthesis. This form of implicit resolution system can be very powerful, but can't be used in conjunction with solver-automated reasoning so, for example, Idris would not be able to handle our sum example which requires automated reasoning about arithmetic. Agda [30] combines unification with a second kind of implicit parameter, *instance arguments* [14], that uses a separate, specialized, implicit resolution mechanism for implicit arguments that relate to typeclasses. Coq [39] uses a mechanism called canonical structures [27], which uses a programmable hint system, but is intimately tied to the specifics of Coq's implementation, and its use for implicit parameter instantiation lacks a formal description, as Devriese et al. [14] lament.

As Devriese et al. [14] note, the complexity of dependent type systems make even achieving similar functionality as in Haskell and Scala a significant task. These implicit parameters are designed to accomplish similar tasks as in the non-dependent Haskell and Scala: automating the instantiation of repetitive arguments or automatically searching the context for relevant arguments. Refinement types allow us to sidestep this issue as the base type system can separately use implicit parameters to automate typeclasses or other programming tasks, allowing our technique of Implicit Refinement Types to entirely focus on using dependent type information. Here, this means focusing on allowing us to use an SMT to simplify ghost specifications without worrying about interactions with the base type system.

While there is much work on implicit parameters for *dependent types*, we believe that we provide the first formal description of a system combining implicit parameters with refinement types. F* is the only other example of a language that has implicit parameters and refinement types, but we discuss how F*'s implicits cannot take advantage of refinement type information in §7.

Horn Constraints. Horn Clauses have emerged as a lingua franca of verification tools [6] as they offer a straightforward encoding of assertions. Z3 [13] includes a fixpoint solver for Horn Constraints including many quantifier elimination heuristics. Rybalchenko et al. [5] present a semi-decision procedure for solving existential Horn clauses using a template-based CEGAR

loop. Cosman and Jhala [10] use NNF Horn constraints to preserve *scope*. We extend this framework to synthesize refinements for implicit program variables that are *existentially* quantified, in addition to the usual *universally* quantified binders.

Unno et al. [41] show that it is sufficient to add one extra parameter for each higher order argument to any given function to achieve complete higher-order verification with first-order refinements. Their treatment utilizes more automation, *e.g.* interpolation and Farkas' lemma, but is limited to arithmetic specifications, precluding programs manipulating data structures like sets and maps, as demonstrated in §7.

References

1 Retrieving paginated results - AWS SDK for java version 2, 2019. URL: `https://docs.aws.amazon.com/sdk-for-java/v2/developer-guide/examples-pagination.html`.

2 Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS 1988), Edinburgh, Scotland, UK, July 5-8, 1988*, pages 165–175. IEEE Computer Society, 1988. `doi:10.1109/LICS.1988.5115`.

3 Andrew W. Appel. Verified software toolchain. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, volume 7226 of *Lecture Notes in Computer Science*, page 2. Springer, 2012. `doi:10.1007/978-3-642-28891-3_2`.

4 Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*, pages 17–32. IEEE Computer Society, 2008. `doi:10.1109/CSF.2008.27`.

5 Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. Solving existentially quantified horn clauses. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 869–882. Springer, 2013. `doi:10.1007/978-3-642-39799-8_61`.

6 Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte, editors, *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015. `doi:10.1007/978-3-319-23534-9_2`.

7 Edwin Brady. State machines all the way down. Draft, 2016. URL: `https://www.idris-lang.org/drafts/sms.pdf`.

8 Edwin C. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013. `doi:10.1017/S095679681300018X`.

9 Adam Chlipala. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, Boston, MA, USA - September 25 - 27, 2013*, pages 391–402. ACM, 2013. `doi:10.1145/2500365.2500592`.

10 Benjamin Cosman and Ranjit Jhala. Local refinement typing. *Proceedings of the ACM on Programming Languages*, 1(ICFP):26:1–26:27, 2017. `doi:10.1145/3110270`.

11 Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 341–360. ACM, 2010. `doi:10.1145/1869459.1869489`.

12 Bruno C. d. S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. The implicit calculus: a new foundation for generic programming. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, Beijing, China - June 11 - 16, 2012*, pages 35–44. ACM, 2012. doi:10.1145/2254064.2254070.

13 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

14 Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in agda. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 143–155. ACM, 2011. doi:10.1145/2034773.2034796.

15 Jana Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, Boston, MA, USA - September 25 - 27, 2013*, pages 429–442. ACM, 2013. doi:10.1145/2500365.2500582.

16 Burak Emir, Andrew Kennedy, Claudio V. Russo, and Dachuan Yu. Variance and generalized constraints for c$^{\#}$ generics. In Dave Thomas, editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 279–303. Springer, 2006. doi:10.1007/11785477_18.

17 Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 341–350. ACM, 2011. doi:10.1145/2046707.2046746.

18 Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 115–130. IEEE Computer Society, 2011. doi:10.1109/SP.2011.36.

19 Martin A. T. Handley, Niki Vazou, and Graham Hutton. Liquidate your assets: reasoning about resource usage in liquid haskell. *Proceedings of the ACM on Programming Languages*, 4(POPL):24:1–24:27, 2020. doi:10.1145/3371092.

20 Christian Heinlein. Implicit and dynamic parameters in C++. In David E. Lightfoot and Clemens A. Szyperski, editors, *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13-15, 2006, Proceedings*, volume 4228 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2006. doi:10.1007/11860990_4.

21 Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer, 2006. doi:10.1007/11691372_33.

22 Ming Kawaguchi, Patrick Maxim Rondon, and Ranjit Jhala. Type-based data structure verification. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 304–315. ACM, 2009. doi:10.1145/1542476.1542510.

23 Kenneth Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software,*

ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings, pages 505–519, 2007. doi:10.1007/978-3-540-71316-6_34.

24 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20.

25 Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark Shields. Implicit parameters: Dynamic scoping with static types. In Mark N. Wegman and Thomas W. Reps, editors, *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, pages 108–118. ACM, 2000. doi:10.1145/325694.325708.

26 Tomer Libal. The unification algorithm, 2017. URL: https://github.com/FStarLang/FStar/wiki/The-unification-algorithm.

27 Assia Mahboubi and Enrico Tassi. Canonical structures for the working coq user. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 19–34, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-39634-2_5.

28 Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. SHILL: A secure shell scripting language. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014, Broomfield, CO, USA, October 6-8, 2014*, pages 183–199. USENIX Association, 2014. doi:10.5555/2685048.268506.

29 Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 229–240. ACM, 2008. doi:10.1145/1411204.1411237.

30 Ulf Norell. Dependently typed programming in agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI 2009: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009. doi:10.1145/1481861.1481862.

31 Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976. doi:10.1007/BF00268134.

32 Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Language and Systems*, 22(1):1–44, 2000. doi:10.1145/345099.345100.

33 Jon Postel. Transmission control protocol. STD 7, RFC Editor, September 1981. URL: http://www.rfc-editor.org/rfc/rfc793.txt.

34 Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169. ACM, 2008. doi:10.1145/1375581.1375602.

35 Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Verifying message-passing programs with dependent behavioural types. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 502–516. ACM, 2019. doi:10.1145/3314221.3322484.

36 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming*, pages 266–278. ACM, 2011. doi:10.1145/2034773.2034811.

37 Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016. doi:10.1145/2837614.2837655.

38 Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, June 16-19, 2013*, pages 387–398. ACM, 2013. doi:10.1145/2491956.2491978.

39 The Coq Development Team. The Coq Reference Manual, 2009.

40 Anish Tondwalkar, Matthew Kolosick, and Ranjit Jhala. Refinements of futures past: Higher-order specification with implicit refinement types (extended version), 2021. arXiv:2105.01954.

41 Hiroshi Unno, Tachio Terauchi, and Naoki Kobayashi. Automating relatively complete verification of higher-order functional programs. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, Rome, Italy - January 23 - 25, 2013*, pages 75–86. ACM, 2013. doi:10.1145/2429069.2429081.

42 Niki Vazou, Alexander Bakst, and Ranjit Jhala. Bounded refinement types. In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 48–61. ACM, 2015. doi:10.1145/2784731.2784745.

43 Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. Abstract refinement types. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2013. doi:10.1007/978-3-642-37036-6_13.

44 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for haskell. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN International Conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 269–282. ACM, 2014. doi:10.1145/2628136.2628161.

45 Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: complete verification with SMT. *Proceedings of the ACM on Programming Languages*, 2(POPL):53:1–53:31, 2018. doi:10.1145/3158141.

46 Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989. doi:10.1145/75277.75283.

47 Leo White, Frédéric Bour, and Jeremy Yallop. Modular implicits. In Oleg Kiselyov and Jacques Garrigue, editors, *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014*, volume 198 of *EPTCS*, pages 22–63, 2014. doi:10.4204/EPTCS.198.2.

48 Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Jack W. Davidson, Keith D. Cooper, and A. Michael Berman, editors, *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 249–257. ACM, 1998. doi:10.1145/277650.277732.

Dealing with Variability in API Misuse Specification

Rodrigo Bonifácio
Computer Science Department, University of Brasília, Brazil

Stefan Krüger
Independent Researcher, Munich, Germany

Krishna Narasimhan
Technical University of Darmstadt, Germany

Eric Bodden
Paderborn University, Germany
Fraunhofer IEM, Paderborn, Germany

Mira Mezini
Technical University of Darmstadt, Germany

Abstract

APIs are the primary mechanism for developers to gain access to externally defined services and tools. However, previous research has revealed API misuses that violate the contract of APIs to be prevalent. Such misuses can have harmful consequences, especially in the context of cryptographic libraries. Various API-misuse detectors have been proposed to address this issue – including CogniCrypt, one of the most versatile of such detectors and that uses a language (CrySL) to specify cryptographic API usage contracts. Nonetheless, existing approaches to detect API misuse had not been designed for systematic reuse, ignoring the fact that different versions of a library, different versions of a platform, and different recommendations/guidelines might introduce variability in the correct usage of an API. Yet, little is known about how such variability impacts the specification of the correct API usage. This paper investigates this question by analyzing the impact of various sources of variability on widely used Java cryptographic libraries (including JCA/JCE, Bouncy Castle, and Google Tink). The results of our investigation show that sources of variability like new versions of the API and security standards significantly impact the specifications. We then use the insights gained from our investigation to motivate an extension to the CrySL language (named MetaCrySL), which builds on meta-programming concepts. We evaluate MetaCrySL by specifying usage rules for a family of Android versions and illustrate that MetaCrySL can model all forms of variability we identified and drastically reduce the size of a family of specifications for the correct usage of cryptographic APIs.

2012 ACM Subject Classification Software and its engineering; Software and its engineering → Domain specific languages; Software and its engineering → API languages; Theory of computation → Cryptographic protocols

Keywords and phrases API misuse, cryptographic API misuse detection, code generation, domain engineering, cryptographic standards

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.19

Funding This research was supported by the DFG's collaborative research center 1119 CROSSING. *Rodrigo Bonifácio*: funded by FAP-DF (research grant 05/2018).

1 Introduction

Application Programming Interfaces (APIs) have become fundamental to increase developer productivity. Nonetheless, prior research [1,15,33] has indicated that developers often struggle with using APIs for various reasons, including poor documentation, low-level abstraction, and lack of tool support. One way to mitigate these issues are approaches to detecting API

misuses [2, 12, 20, 24, 27]. Most of these approaches are deny-listing approaches [2, 12, 20, 27] – providing analyses that scan for *incorrect* API uses. Deny-listing approaches suffer from false negatives and cannot be easily extended because they rely on hard-coded rules [24]. To address these issues, CogniCrypt [24] follows an allow-list approach and instead of hard-coding what the correct usages are, it takes a set of correct usage rules as a parameter – the latter are specified by API developers using the CrySL specification language [24]. For instance, CrySL has been used to model correct usage rules of Java Cryptographic APIs.

However, the correct usage of an API is often subject to various sources of variability. They include (but are not limited to) evolving signatures and behavioral changes, e.g., due to different security standards in case of crypto APIs.[1] Last but not least, APIs like Java Cryptography Architecture (JCA) foster flexibility through the use of different *providers* that can be *plugged into* to override the default implementation of an algorithm. Depending on the JCA provider, different secure algorithms (according to cryptographic standards) might be available or not. Whether or not an API usage is correct may also vary owing to other factors, including version of the platform (e.g., Java Platform, Android Platform) and version of the API implementations. Nonetheless, there is a lack of understanding about (a) how sources of API variability affect what should be considered the correct usage of an API and (b) a solution to modelling this variability in allow-listing approaches like CrySL. This is where this paper makes its contributions.

We perform an in-depth domain engineering on the correct usage of cryptographic APIs. To this end, we consider the following sources from which variability might originate from: cryptographic standards (FIPS, ECrypt, and BSI), cryptographic libraries (e.g., JCA, Google Tink, Bouncy Castle), cryptographic library implementations (e.g., JCA providers), and cryptographic library evolution. Based on the findings, we implement Meta-CrySL, a meta-programming approach for managing families of CrySL specifications, ensuring that different sources of variability can be accounted for when specifying usage patterns. Using the new set of specifications, we conduct an empirical study to investigate two characteristics of Meta-CrySL: *expressiveness* (i.e., *the possibility to express all sources of CrySL variability using Meta-CrySL*), *compactness* (i.e., *number of lines of CrySL code one can save when writing Meta-CrySL specifications* and the fraction of redundancy one can eliminate) and *correctness* (i.e., *does the specifications generated by Meta-CrySL detect distinct violations when exploring different configurations of CrySL rules*).

We believe that one can also benefit from using a domain engineering approach for specifying the correct usage of non-cryptographic APIs as well. First because the sources of variability we discuss are not unique to cryptographic APIs as all APIs offer variability in behavior due to evolving signatures as a result of new versions. Second because variability as a result of pluggable implementations from different providers is not unique to JCA, either (c.f., JDBC [2]). Even security standards that are unique to cryptographic APIs have parallels in the form of context-specific usage patterns for non-crypto APIs.

To summarize, the main contributions of this paper are as follows:

- Domain engineering on Java cryptographic libraries, including:
  - A study on the evolution of Java cryptographic APIs.
  - A study on different cryptographic standard recommendations.
  - A discussion about how the evolution of cryptographic libraries and cryptographic recommendation impact on the correct usage of APIs.

[1] Cryptographic libraries have different definitions of correctness – -and in particular *secure* – usages, based on the standards like FIPS or BSI under which they operate contributing to yet another source of variability.

[2] https://www.oracle.com/database/technologies/jdbc-migration.html

- The design and implementation of Meta-CrySL, an extension to CrySL that helps manage sources of variability on CrySL specifications.
- An evaluation that shows how Meta-CrySL can help API experts to better modularize variability in CrySL specifications.

In Section 2, we discuss some concepts that are pre-requisite to understanding the remainder of the paper. We present our analysis of sources of API variability in Section 3. In Section 4, we present the design of Meta-CrySL, the language that resulted out of the insights gained from our study. Lastly, we empirically evaluate Meta-CrySL in Section 5.

2 Background

In this section, we present the concepts and definitions necessary to understand our research context, contributions, and results. In Section 2.1, we introduce the challenges for using cryptographic APIs correctly. Although we perform the first part of our study with different cryptographic APIs, we will use the JCA to drive home these challenges in this section. In Section 2.2, we present the cryptographic standards we consider in our research. These standards may guide and impact the specifications of the correct usage of cryptographic APIs. Finally, Section 2.3 introduces the CrySL language, which allows experts to specify the proper usage of Java cryptographic APIs.

2.1 Cryptographic APIs

Ferguson et al. [14] state that "cryptography is very difficult", mostly because it involves several branches of mathematics and computer science [14, 44]. For this reason, algorithms and implementations are only recommended after a huge effort on testing – often conducted by a public community. That is, regardless of how much they have been vetted, they are at best *still secure* or *not yet insecure*. As a result, developers should rely on well-known cryptographic algorithms and API implementations that are subject to hundreds or thousands of hours of cryptanalysis [44].

Cryptographic APIs (or libraries) that exist for each major programming language,such as JCA and Bouncy Castle for Java and wolfCrypt and OpenSSL for C/C++, make available a number of implementations for performing cryptographic tasks, such as the support for generating (pseudo) random numbers, message digests, symmetric and asymmetric cryptography (including digital signature). Although these libraries share similar characteristics, their design differ according to distinct principles, such as flexibility and usability. Unfortunately, existing research reports that these APIs are often complex and hard to use [1, 33], which in the end might compromise the security of the systems.

For instance, JCA has been designed such that it is possible to change the cryptographic implementations used in a system without having to modify many parts of the system. Specifically, this API employs the provider architecture [21] that enables implementations behind the interfaces to be easily swapped. The official documentation of JCA [21] explicitly mentions that the three main motivations driving the design of the API were:

1. **Implementation independence:** Applications can choose between many variants of implementations of cryptographic algorithms
2. **Implementation interoperability:** Just like the applications are not tied to providers, providers are also not tied to applications
3. **Algorithm extensibility:** Cryptographic algorithms can use building block primitives from variable sources to compose their algorithms

Figure 1 shows a usage scenario for the `MessageDigest` class of the JCA, which computes a hash of input data. The first step to this end is to get an instance of an implementation using a string that specifies the message digest algorithm (`BLAKE2B-512`), and, optionally, a named reference to a provider that makes available the actual implementation of the algorithms through the JCA interface. After getting a `MessageDigest` instance, a developer might populate the digest by calling the `update()` method one or more times, and then calling the `digest` method to compute a hash value of the input data. The same sequence of events has been valid since the first specification of this API. However, several new message digest algorithms have been implemented (e.g., the family of SHA-3 algorithms has been introduced in Java 9). Others have been deprecated and considered insecure over the years (e.g., algorithms MD2, MD5, and SHA-1 are not recommended anymore [16]).

```
@Test
public void testBlakeDigest() {
  try {
    MessageDigest md = MessageDigest.getInstance("BLAKE2B-512", "BC");
    md.update(data);
    byte[] res = md.digest();
    Assert.assertNotNull(res);
  }
  catch(Exception e) {
    org.junit.Assert.fail(e.getMessage());
  }
}
```

Figure 1 Code snippet for computing a message digest using the JCA Bouncy Castle provider (identified by the `BC` string).

Therefore, to correctly use JCA, developers must not only understand the expected sequence of method calls for each cryptographic primitive, but which algorithms and providers are available and are still considered secure. Cryptographic standards detail which algorithms and algorithm configurations developers should use while implementing systems that deal with sensitive information. Given the complexity related to the use of crypto APIs, existing research uses static analysis tools to assess the correct usage of crypto APIs [22, 24, 38] and code generation to assist developers to correctly implement cryptographic tasks [25].

2.2 Cryptographic standards

A cryptographic standard details a set of recommendations related to the use of cryptographic primitives. A few examples of cryptographic standards include:

FIPS Standards present a set of requirements from the American National Institute of Standards and Technology (NIST) that should be considered when implementing security modules for computational systems [34]. This set of standards suggest algorithms for different primitives, including symmetric encryption, digital signatures, and message digest.

BSI TR-021-102-1 is a technical guideline from the German Federal Office for Information Security (BSI) that provides the results of a security assessment on cryptographic algorithms. This assessment supports a long-term orientation on the use of cryptographic mechanisms [16].

ECrypt TR-D5.4 details a set of recommendations about cryptographic algorithms and key size. It is an effort from the Ecrypt Coordination and Support Action, an initiative from the European Unions' H2020 program [13].

2.3 CrySL: Assessing the Correct Usage of Cryptographic APIs

As can be seen from the above, applying cryptographic APIs within a software can have a lot of potential for errors and developers require new techniques and tools to support the use of cryptographic APIs. A research effort involving different institutions designed and developed CogniCrypt, a suite of tools that leverages the specification language CrySL to enable API experts to specify the correct usage of libraries. CogniCrypt$_{SAST}$ [24] is a module of CogniCrypt that takes rules in CrySL and a target program as input and uses state-of-the-art data-flow analysis [45–47] to identify deviations from these rules in this program.

In its current version, CrySL allows cryptographic experts to specify how to instantiate and use an object-oriented class that implements a cryptographic primitive. Figure 2 shows the CrySL specification for the `MessageDigest` class of the JCA API.

```
SPEC java.security.MessageDigest
OBJECTS
  java.lang.String algorithm;
  byte[] data;
  byte[] digest;
EVENTS
    g1: getInstance(algorithm);
    g2: getInstance(digestAlg, _);

    Gets := g1 | g2;

    u1: update(_);
    d1: out = digest();
ORDER
    Gets, u1+, d1
CONSTRAINTS
    algorithm in {"SHA-256", "SHA-384", "SHA-512", "BLAKE2B-512"};

ENSURES
    digested[out];
```

Figure 2 CrySL rule for the `MessageDigest` JCA API (considering the default provider).

A CrySL rule explicitly states the *class under specification* in the `SPEC` clause. The `OBJECTS` definition describes a list of object declarations. These objects might appear as arguments to events or as variables assigned to the return value of an event. The `EVENTS` section declares the methods of the *class under specification* that are relevant for specifying the correct usage of the class. In particular, the order in which these (labeled) methods should be called appears as a regular expression in the `ORDER` clause. Several operators can be used to denote this regular expression. That is, supposing that we have events with labels e_1 and e_2, we can combine these events using either the *sequence operator* (e_1, e_2) or the "or" operator $(e_1 \mid e_2)$. We can also state that one event is optional $(e_1?)$ or that an event might either occur zero or more times (e_1*) or one or more times (e_1+). It is also possible to define *aggregates* (such as `Gets := g1 | g2`), which help with the definition of the `ORDER` clause. The example of Figure 2 states that a developer must first call one of the `getInstance()` methods (using the `Gets` aggregate) before calling the `update()` method at least once. After that, the developer must conclude the computation of the message digest using the `digest()` method. A CrySL compiler translates this regular expression into a state machine. After that, the CogniCrypt$_{SAST}$ component [24] analyzes a system to verify if a sequence of calls to a `MessageDigest` instance obeys the expected sequence of events of the `ORDER` clause.

The `CONSTRAINTS` clause allows a cryptographic expert to define constraints on the objects declared in a CrySL rule. For instance, the CrySL rule of Figure 2 states that the `algorithm` used as parameter for the `getInstance()` methods should be evaluated to one of the string literals that represent a "secure" message digest algorithm supported by the JCA default providers: `SHA-256`, `SHA-384`, or `SHA-512`. Therefore, during the analysis of a system, CogniCrypt$_{SAST}$ reports an error if it finds a call to the `getInstance()` method of the `MessageDigest` class using a different algorithm (such as `MD5`). Finally, the `ENSURES` clause of a CrySL rule allows a cryptographic expert to state a predicate that can be later used as a pre-condition in a CrySL specification for a different class (using the `REQUIRES` construct of CrySL). There are other CrySL constructs that we do not discuss here, and a reader that is interested in a more detailed description should read the paper that introduced the CrySL specification language [24].

Previous studies have shown the efficiency of using the CrySL approach in identifying common misuses of cryptographic APIs [24], *but considering only one specific set of CrySL rules*. Nonetheless, as we discuss in the remainder of this paper, CrySL rules should consider possible sources of variability that might affect the specifications, including versions of APIs and platforms and cryptographic standards.

3 Domain Analysis

To better understand the impacts of variability on API misuse specification, we conducted a *domain analysis* [4, 36] that sought to understand reuse opportunities across Crypto-API-usage specifications, considering different libraries, their different providers and their different versions, different cryptographic primitives, and different cryptographic standards – altogether corresponding to the *sources of variability* of our domain analysis. Domain Analysis is a well-established set of activities in the software product line community. The goal is to identify variability motivating the implementation of an infrastructure for software reuse [4, 36].

3.1 Study Settings

We setup our study based on the following research questions:

RQ1 **How do different APIs and their implementations (e.g., different JCA providers) vary the specifications of the correct usage of cryptographic primitives?** Motivation: Previous studies using specification languages like CrySL only considered the correct usage of the *default* providers for the JCA. These studies report that almost 95% of Android applications that use cryptographic APIs present at least one misuse of these APIs [24]. Answering **RQ1** is relevant because alternative providers such as Bouncy Castle support algorithms that are not supported by the default providers. It is unclear whether findings of the previous CrySL studies remain valid (particularly in the cases where an application explicitly uses a different provider).

RQ2 **How do existing cryptographic standards vary the notion of secure or compliant use of cryptographic libraries?** Motivation: Although the use of some cryptographic algorithms are considered insecure (e.g., `MD5` and `SHA-1`), they are still widely used in practice. There are many reasons for that, including compatibility with existing legacy code and the lack of knowledge of developers about up-to-date cryptographic algorithm recommendations. In addition, current security standards (such as FIPS and ECrypt) present recommendations about which algorithms should be used now and in a near future. Answering **RQ2** helps us to construct a baseline regarding

how secure existing application are when considering existing standards. Moreover, **RQ2** helps to understand the relevance of security standards to the specification of the correct usage of cryptographic APIs.

RQ3 **How does the evolution of a cryptographic library vary its correct usage over time?** Motivation: Answers to **RQ3** will bring new insights about how to specify policies and static analysis tools that aim to guarantee the correct usage of cryptographic APIs, considering that they might evolve along the way. Moreover, answering **RQ3** might provide evidence that the evolution of APIs must be considered when specifying their correct usage.

To answer the RQs, we first conducted a domain analysis on the specification of the correct usage of Cryptographic APIs. We first read the documentation of APIs and looked at code examples (including test cases) that use Java (JCA, Bouncy Castle, and Google Tink) and C/C++ (OpenSSL and wolfCrypt) cryptographic libraries. We then built a general understanding about how different sources of variability might influence our domain, i.e., the domain of specification of the correct usage of cryptographic APIs.

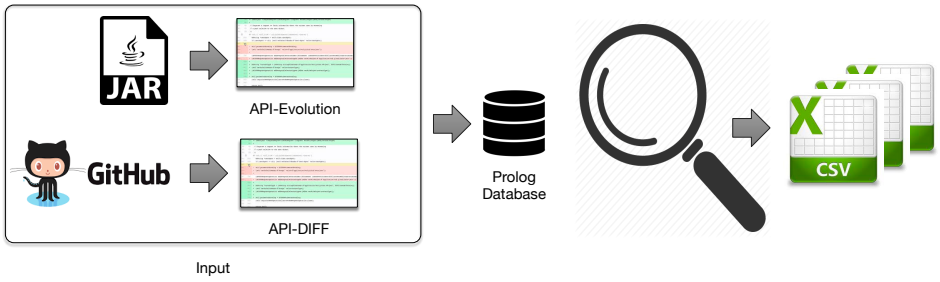


Figure 3 Approach for mining the evolution of Java Cryptographic APIs.

Figure 3 shows the general workflow that we use to mine the evolution of Java cryptographic libraries (JCA, Google Tink, and Bouncy Castle). We leverage APIDIFF [7, 8] and our own static analysis tool to mine classes and methods available per API release and the patterns of changes along the evolution of the libraries. We populate all this information into a Prolog database of facts and rules that allow us to answer questions concerning both newly introduced algorithms as well as deprecated ones for specific versions of a given library. Introducing and removing new primitive algorithms suggest that there should exist CrySL rules for every version of that API that introduces a change. Using a customized version of APIDIFF, we also investigated breaking changes [6, 8, 49], that is, changes between consecutive releases of an API that break the client code. Next, we execute queries into this database and export the results to CSV files to analyze and understand the evolution of crypto APIs.

3.2 Analysis Results

RQ1: Variability Due to Different Cryptographic APIs

We started our domain analysis by exploring different cryptographic APIs (e.g., JCA, Google Tink, and Bouncy Castle, Open SSL and wolfCrypt). We soon realized that these APIs differ significantly in terms of design principles and decisions. For instance, the design of the JCA considers flexibility as a key element. Developers are responsible for specifying the configurations of keys and algorithms as well as modes of operations they want to use,

```
byte[64] digestData(byte input[64]) {
  byte digest[64];
  Blake2b b2b;
  wc_InitBlake2b(&b2b, 64);
  wc_Blake2bUpdate(&b2b, input, sizeof(input));
  wc_Blake2bFinal(&b2b, digest, 64);
  return digest;
}
```

Figure 4 Function for hashing a byte array using Blake2b.

which has proven to be challenging to developers. Certain configuration problems might only appear at runtime adding to the complication. The design decisions of Google Tink, on the other hand, favor simplicity, instead of flexibility. This way, there is a small set of key / algorithm configurations available, and the developer is encouraged to use one of these configurations, in order to avoid possible API misuses.

In comparison to Google Tink, C/C++ libraries are yet more restricted. That is, OpenSSL and wolfCrypt define specific functions for each algorithm. The code snippet in Figure 4 shows how to use the wolfCrypt library to generate a hash of an input data using the Blake2b algorithm. There are several calls to functions that are specific to this algorithm. Since the different implementations of message digest algorithms in wolfCrypt do not share a common interface, the code of the `digestData` function is not flexible. In the case a developer has to change the message digest algorithm, she would have to rewrite the entire function.

Based on our analysis of the different APIs, we understand that it is difficult to reuse usage rules across different APIs and languages. Nonetheless, we found some opportunities to reuse CrySL rules across different JCA providers and within the Google Tink and Bouncy Castle libraries. These opportunities mostly arise due to existing security standard recommendations (we can customize the specifications that address either FIPS or ECrypt recommendations, for instance), due to the evolution of the API implementations, and due to the similarity we found among different primitives and primitives' implementations. We present some examples of these situations in the remainder of this section.

> There is no clear opportunity for reusing the specifications of the correct usage of cryptographic libraries across different APIs and languages.

RQ2: Variability Due to Cryptographic Standards

Existing technical reports and standards present a series of recommendations about which cryptographic algorithms (and respective key configurations) should be used in applications. These technical reports characterize a valuable source of information to indicate whether a given system is "secure according to a given standard". Moreover, (some) existing cryptographic APIs (e.g., wolfCrypt and Bouncy Castle) comply to the FIPS certifications – and using a certified library according to the standard recommendations might represent a competitive advantage for products in specific domains. For instance, FIPS 140-2 validation is mandatory for use in the US Federal systems that collect or store sensitive information.[3]

We found that existing standards introduce a source of variability in usage specifications. This source of variability occurs because sets of algorithms (and algorithm modes) are recommended by some standards, but not in others. Message digests represent one point

[3] https://csrc.nist.rip/groups/STM/cmvp/

in variability as Table 1 shows. All standards mentioned in Section 2.2 specify secure hash algorithms that may be used to process a message and produce a condensed representation (a message digest). However, they do not all recommend the same.

Table 1 Recommendations for using different message digest algorithms.

Algorithm	FIPS	BSI	ECrypt
MD5	✗	✗	✗
SHA-1	✗	✗	✗
SHA-224	✓	✗	✗
SHA-256, 384, 512	✓	✓	✓
SHA-512/224	✓	✗	✗
SHA-512/256	✓	✓	✓
SHA-3/(256, 384, 512)	✓	✓	✓
Shake128, Shake256	✓	✓	✓
Whirlpool	✗	✗	✓
Blake	✗	✗	✓

If we were to encode these standards in CrySL, we would need to model them in three distinct rules that nonetheless largely overlap. Let us discuss these rules in more detail. First, consider the default CrySL specification for the `MessageDigest` class of the JCA, when considering the default provider (Figure 2). In this case, the set of supported algorithms on Line 17 is limited to the default algorithms of JCA.

In case we specify aforementioned standards, we would have to consider using the Bouncy Castle JCA provider – since the default provider does not support some of the algorithms in Table 1 (such as Whirlpool and Blake), and change that line to consider the recommended algorithms of each standard, as we show in Figures 5. In this particular case, it is possible to reuse almost all the CrySL specification of Figure 2, changing only the `algorithm` constraint based on the supported standard / technical report. We name this kind of variability *Variability on Set Constraints*.

Bouncy Castle provides a lightweight API on top of the providers for JCA [4]. Considering the Lightweight Bouncy Castle API, one is required to write a CrySL rule for each primitive implementation, as shown in Figure 6 for SHA256 and SHA512. Instead of one specification for each cryptographic standard (varying the supported algorithms), there are several CrySL rules for each cryptographic standard (one per supported primitive implementation). The variability here relates to the classes that implement the message digest primitives and the Lightweight Bouncy Castle API implements the individual algorithms in a distinct class. However, the corresponding CrySL specifications vary only according to the base class (in the example, `SHA256Digest` and `SHA512Digest`). We name this kind of variability *Variability on the Base Specification Class*.

> The specification of the correct usage of cryptographic APIs should consider the recommendations of individual cryptographic standards. The impact on the specifications due to a cryptographic standard depends on the API.

[4] `https://www.bouncycastle.org/`

```
SPEC java.security.MessageDigest
// same definitions of the default JCA MessageDigest specification
CONSTRAINTS
  algorithm in {"SHA-224", "SHA-256", "SHA-384", "SHA-512", "SHA-3", "Shake-128", "Shake-256"};
ENSURES
    digested[out];
```

(a)

```
SPEC java.security.MessageDigest
// same definitions of the default JCA MessageDigest specification
CONSTRAINTS
  algorithm in {"SHA-256", "SHA-384", "SHA-512", "SHA-3", "Shake-128", "Shake-256"};
ENSURES
    digested[out];
```

(b)

```
SPEC java.security.MessageDigest
// same definitions of the default JCA MessageDigest specification
CONSTRAINTS
  algorithm in {"SHA-256", "SHA-384", "SHA-512", "SHA-3", "Shake-128", "Shake-256", "Whirlpool",
                "Blake2s", "Blake2b"};
ENSURES
    digested[out];
```

(c)

Figure 5 CrySL rules for the `MessageDigest` JCA (considering the Bouncy Castle provider and the (a) FIPS recommended algorithms, (b) BSI recommended algorithms, and (c) ECrypt recommended algorithms).

RQ3: Variability Due to the Evolution of the APIs

We conduct this study using the approach introduced in Section 3.1, to identify cryptographic algorithms introduced/removed and in turn the breaking changes between two public releases of an API. In this case we considered three APIs: JCA, Lightweight Bouncy Castle, and Google Tink. These APIs already have CrySL specifications for them.

Specifically, we mine the evolution history of 15 releases of the Lightweight Bouncy Castle (v.1.46 to v.1.60), all available in the Maven Central Repository.[5] Later we summarize some findings related to the evolution of the Google Tink and JCA.

The classes that implement the cryptographic primitives in Lightweight Bouncy Castle implement one of the existing interfaces declared in the Java package `org.bouncycastle.crypto`, including the `Digest`, `Mac`, and `BlockCipher` interfaces. In the last Bouncy Castle release considered in our analysis (release 1.60), we identified more than 140 primitive implementations, among them 45 implementations of the `BlockCipher` interface.[6] Block cipher (45), message digest (29), message authentication code (18), and stream cipher (21) are the primitives with the most algorithm implementations.

Figure 7 shows the evolution in the number of implementations for these primitives. We can see that almost all releases introduce at least one new primitive implementation. For instance, release 1.47 introduced a new implementation of the Mac primitive, while release

[5] `https://search.maven.org/`

[6] We analyzed these `BlockCipher` implementations and we found classes that implement cipher algorithms (e.g., AES and Blowfish) and cipher modes (e.g., CBC and GCM).

```
SPEC SHA256Digest

OBJECTS
  byte input;
  byte[] out;
  int outOff;

EVENTS
  c : SHA256Digest();
  u : update(input);
  f : doFinal(out, outOff);

ORDER
  c, u+, f

ENSURES
  digested[out];
```

(a)

```
SPEC SHA512Digest

OBJECTS
  byte input;
  byte[] out;
  int outOff;

EVENTS
  c : SHA512Digest();
  u : update(input);
  f : doFinal(out, outOff);

ORDER
  c, u+, f

ENSURES
  digested[out];
```

(b)

Figure 6 Specification of CrySL rules for the message digest classes in the Bouncy Castle lightweight API. We will have to elaborate one specification for each supported algorithm of a standard / technical report.

1.59 introduced five new block ciphers, one new message digest, and three new stream ciphers. Only releases 1.52, 1.56, and 1.60 did not introduce any new such primitive.

The existence of different implementations of a given primitive has an influence on the specifications of the correct usage of an API. Consider again the test case method on Figure 1. This example uses the Bouncy Castle JCA provider (named `"BC"`) for generating a digest of an input data using the *Blake2b* algorithm. However, this algorithm was first introduced in the release 1.53 of Bouncy Castle. If one executes this test case using an earlier release (e.g., 1.51 or 1.52), the test case fails with a `NoSuchAlgorithmException`. Therefore, the CrySL specification of Figure 5(c) is not compliant with the releases of Bouncy Castle prior to 1.53.

> What is considered correct usage of an API depends on the specific versions of the API.

We also analyzed the changes in the Bouncy Castle API that might cause an undesired effect on the client systems [49] and identified *breaking changes*. Breaking changes include *removing a public method*, *renaming a public method*, and *reducing visibility of a method*. A catalog of these changes could be found elsewhere [11]. We only consider the twelve releases from 1.49 until 1.60 because these releases are available in the public Git source code repository of Bouncy Castle.

Using the same approach of previous works [7,8,49], we found 1.733 scenarios of breaking changes – considering all pairs of successive releases. We document them all in Figure 8. In total, we identified 1.162 removals of public methods (67% of all breaking changes). All releases feature at least one. Similarly, all releases change the return type of at least one method. There are a total of 128 occurrences. Other common breaking changes are *change in exception list* (172 cases), *renaming a public method* (130 cases), and *reducing visibility of a method* (92 cases). The remaining 49 breaking fall into other categories. Four pairs of successive releases contribute with 74.49% of the breaking changes: 1.58–1.57 (210 cases), 1.57–1.56 (364 cases), 1.51–1.50 (389 cases), and 1.50–1.49 (328 cases). We did not find any evidence that one specific release accounts for a major redesign of the Bouncy Castle library.

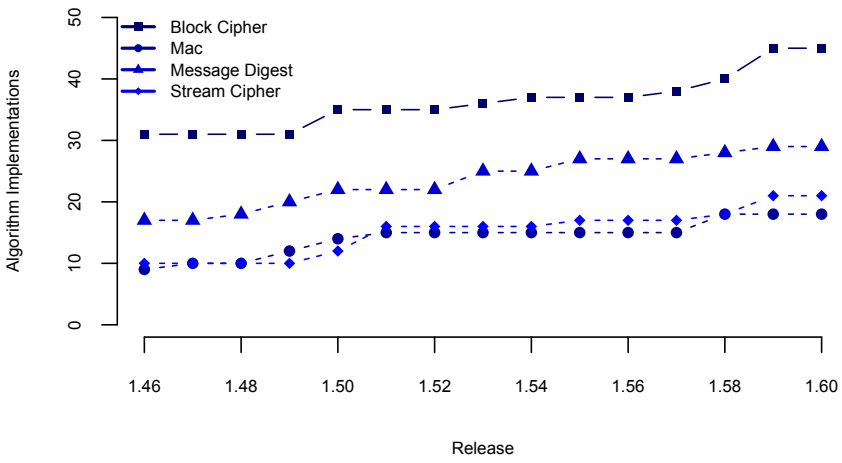


Figure 7 Evolution in the number of algorithm implementations in Bouncy Castle.

Based on these numbers, one might conclude that, despite its long history, Bouncy Castle is an unstable library. This conclusion would be true if developers depended on the public interfaces of the classes that present breaking changes. However, when we consider only the Java interfaces that define the contract of cryptographic primitives (such as the `Digest` and `BlockCipher` interfaces), we found that the Bouncy Castle library is quite stable. Considering all releases, we only identified 34 breaking changes (12 occurrences of *removing a public method*, 9 occurrences of *changing the return type of a public method*, 7 occurrences of *renaming a public method*, 4 occurrences of *reducing visibility of a method*, and 2 occurrences of *changing the exception list of a method*). Yet, we do not know whether or not developers only rely on these "high level" interfaces.

> We found 1.733 breaking changes along 11 public releases of Bouncy Castles. However, considering the core interfaces of the library, we only found 34 breaking changes that might also induce changes on CrySL specifications.

Method updates like renaming/removing/adding methods requires changes to the event section of CrySL specifications. We name this variability as *Variability on Event Sets*.

We further investigated whether the Google Tink library is more stable: the public interfaces of the classes are almost unchanged between the release 1.0.0 (published in September 2017) and the release 1.2.1 (published in November 2018). During this period, we found 50 breaking changes – 43 from version 1.0.0 to version 1.1.0, which might indicate a slight revision on the first design of the library. Between these first initial releases, we identified 14 removed methods. Nevertheless, the most critical change regarding CrySL specifications was the introduction of the Deterministic AEAD algorithm on version 1.1.0. This type of variability is modular and involves only the selection of a set of CrySL specifications (hereafter referred to as *Modular Selection of CrySL Rule*). Although not common, we also identified some variability due to the key templates available across the different versions of Google Tink. The current CrySL specifications for Google Tink can deal with the introduction of key templates that modify the events in the specifications (using the *Variability on Event Sets* strategy).

Figure 8 Total number of breaking changes in the Bouncy Castle API.

Finally, we also considered the evolution of JCA from Java 4 to Java 9. To this end, we analyzed the classes related to the cryptographic primitives available in three standard Java libraries: `rt.jar`, `jce.jar`, and `sunjce_provider.jar`, considering official releases of the Java language. This API is highly stable as it is based on an official Java specification. For instance, the public class interfaces of the JCA do not present any breaking change, and from Java 5 (2005) to Java 9 (2017) the interface of the `java.security.MessageDigest` class did not change. In Java 7, three additional methods that can be used for ciphering a text with *additional authentication data* (AAD) were introduced in the class `javax.crypto.Cipher`. Although the APIs are stable, new primitive algorithms have been introduced along these versions. For instance, eight new ciphers and six new MAC algorithms have been introduced in the JCA, from Java 4 to Java 9.

4 Meta-CrySL

4.1 Design and Implementation Procedures

We used the outcomes of our domain analysis to design and implement Meta-CrySL. Meta-CrySL provides means for the systematic reuse of CrySL specifications. To this end, Meta-CrySL allows the specification of CrySL rules enriched with variation points (such as meta-variables and type parameters) and **refinement** operations that solve these variation points for a given **configuration** (e.g., version of an API or platform, security standard, and so on). Meta-CrySL generates a set of CrySL rules tailored for a given configuration.

We implemented Meta-CrySL using Rascal-MPL [23]. One of the main design decisions was to implement three distinct languages: one for abstract CrySL specifications (i.e., CrySL with variation points), one for CrySL refinements, and one for representing a configuration model. The configuration model states a set of abstract CrySL specifications and refinements. We use a program-generator approach to combine instances of the refinement and configuration languages, and to output regular CrySL specifications. These regular specifications can directly be used with CogniCrypt's infrastructure for CrySL specifications. The following set of high-level requirements guided the design of Meta-CrySL.

1. Meta-CrySL follows a *meta-programming* approach: we write Meta-CrySL specifications and generate regular CrySL specifications from them. Using this design allows us to preserve all CogniCrypt$_{SAST}$ infrastructure.
2. Meta-CrySL should support the sources of variability discussed in the previous section, so that we can generate CrySL rules for different standards and versions of the APIs.

3. Meta-CrySL should also favor reuse among specifications of the same API, reducing the effort in the case that an API supports many algorithms (as for instance Bouncy Castle).

4.2 High-level Architecture

Figure 9 shows the architecture of Meta-CrySL, which follows a multi-staged pipeline for language processing [35], where a module loads a Meta-CrySL `configuration` that specifies a set of extended CrySL specifications and a set of `refinements` that should be used during the building process of a specific set of CrySL rules. After that, the *Loader* module parses the sets of extended CrySL and refinement files, generating abstract representations of these languages as instances of Rascal-MPL algebraic data types (in the following sections we detail these languages). The *Preprocessor* module manipulates these instances executing the refinement operations, using visitors for program transformations. That is, the *Preprocessor* solves Meta-CrySL variability and generates an abstract representation of CrySL rules. Finally, the *Pretty Printer* module outputs regular CrySL specification files.

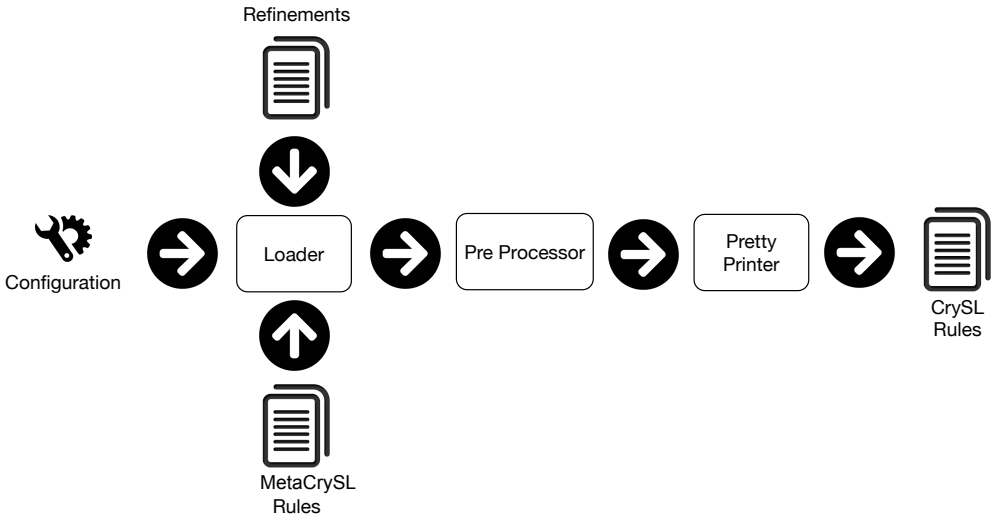


Figure 9 High-level architecture of Meta-CrySL.

4.3 Abstract CrySL Language

Abstract CrySL is an extension of the CrySL language that allows cryptographic specialists to write variation points on the CrySL rules, for instance, in terms of *meta-variables* and type parameters. Figure 10 shows an example of an instance of the abstract CrySL language, modelling variability on CrySL rules for the JCA `MessageDigest` class. The main source of variability in this case relates to the sets of algorithms that might change due to a specific standard or version of the provider implementation (recall the specifications in Figure 5). The abstract CrySL rule of Figure 10 introduces the concept of *meta-variables*, which are bound during the derivation process of CrySL rules. In the example, we can bind the meta-variable `$AlgSet` to the sets of algorithms supported by a given standard (e.g., FIPS, EuroCrypt, or BSI) or specific version of an API.

To deal with *Variability on the Base Specification Class*, we use template-based type parameters, similarly to the mechanism of type expansion supported by C++ templates. As such, when solving this type of variability, we actually generate different copies of a CrySL rule, one for each concrete type that appears in the refinement specifications.

```
SPEC java.security.MessageDigest
// same definitions of the default JCA MessageDigest specification
CONSTRAINTS
  algorithm in $AlgSet;
ENSURES
    digested[out];
```

Figure 10 Use of meta-variables to deal with *Variability on Set Constraints*.

Using the abstract specification of Figure 11, we generated six CrySL rules for different Google Tink primitive's implementations. We also used the same strategy to factor out existing CrySL rules for the message digest primitive of the Lightweight Bouncy Castle API. Each one of these CrySL rules has about 40 lines of code. Using Meta-CrySL, we were able to specify each variant using 4 lines of code (three lines of refinements and one line of the configuration language).

```
ABSTRACT SPEC AbstractFactory<T>
OBJECTS
  com.google.crypto.tink.KeysetHandle ksh;
  <T> primitive;
EVENTS
  gp : primitive = getPrimitive(ksh);
ORDER
  gp
REQUIRES
  generatedKeySet[ksh];
ENSURES
  setPrimitive[primitive];
```

Figure 11 Use of type parameters to deal with *Variability on the Base Specification Class*.

4.4 Refinement Language

Our refinement language allows cryptographic experts to specify transformations on the Meta-CrySL rules, to solve variation points. Considering the discussion of the previous section, Meta-CrySL supports two types of syntactic variation points: meta-variables and type parameters. In addition, it is also possible to introduce new events (and events aggregates), to introduce new constraints, and to replace the events' order of a Meta-CrySL specification. The refinement language expects a base specification and a list of refinement elements.

The current implementation of Meta-CrySL supports different refinement transformations, for instance, transformations that support the kinds of variability discussed in the previous section:

- **Define literal set** binds a meta-variable to a literal set, such as `SHA512, Blake2b, Blake2s`. We use this transformation to solve *Variability on Set Constraints*.
- **Define qualified type** binds a fully qualified type to a type parameter of a Meta-CrySL specification. This transformation solves *Variability on the Base Specification Class*.
- **Add new event** introduces a new event into a Meta-CrySL specification. We use this transformation to solve *Variability on Event Sets*. Similarly, the refinement language also supports operations to add (remove or update) constraints and requires/ensures clauses.

```
SPEC MessageDigest REFINES java.security.MessageDigest {
  define AlgSet = {"Blake2s", "Blake2b", "GOST-3411", "SHA-256", "SHA-384",
                   "SHA-512", "Whirlpool"};
}
SPEC KeyGenerator REFINES javax.crypto.KeyGenerator {
  define AlgSet = {"AES", "BLOWFISH", "HmacSHA256", "HmacSHA384", "HmacSHA512",
                   "RIJNDAEL", "Serpent"};
  add constraint alg in {"AES"} => keySize in {128, 192, 256};
}
```

Figure 12 Example of refinement specifications for the Bouncy Castle JCA Provider.

```
SPEC SHA256 REFINES
     Digest<org.bouncycastle.crypto.digests.SHA256Digest>;
SPEC SHA384 REFINES
     Digest<org.bouncycastle.crypto.digests.SHA384Digest>;
SPEC SHA512 REFINES
     Digest<org.bouncycastle.crypto.digests.SHA512Digest>;
SPEC SHA512t REFINES
     Digest<org.bouncycastle.crypto.digests.SHA512tDigest>;
```

Figure 13 Example of refinements that bind a type parameter for the set of message digest specifications for the Lightweight Bouncy Castle API.

Figure 12 shows two examples of *refinement specifications*. The first *refines* the `MessageDigest` CrySL specification of Figure 10, binding the meta-variable `AlgSet` to a set of message digest algorithms supported by the Bouncy Castle JCA provider. The second refinement specification `KeyGenerator` also defines a set of algorithms supported by the Bouncy Castle JCA provider and also introduces a new constraint which refers to two variables of the base specification (not illustrated in this paper): `alg` and `keySize`. The constraint states that if $alg = AES$, the variable `keySize` must be a value in the set $\{128, 192, 256\}$.

Figure 13 shows a set of refinement specifications that are used for generating CrySL rules for different message digest algorithms supported by the Lightweight Bouncy Castle API. Each refinement specification generates a different CrySL specification, binding a type parameter with the full qualified name of a class that implements a message digest algorithm. In this scenario, we are able to solve all variability using only type parameters, and thus the body of the refinement specifications is empty.

It might be necessary to add further refinement transformations in the future. To implement a new transformation, one would have to modify three Rascal-MPL modules, being necessary to specify the concrete and abstract syntax of the transformation in the refinement language and to implement a new function with the expected behavior of the transformation (`Preprocessor` module). In case one needs to introduce a new syntactic CrySL variation point, this is possible by modifying the abstract and concrete syntax of the abstract CrySL language. We have already implemented six transformations, each one having around ten lines of code.

4.5 Meta-CrySL Configurations

We use a configuration language to specify the Meta-CrySL *building process*. Figure 14 shows an example, which states the base path where the Meta-CrySL implementation could find the specifications and refinements (Line 2), the output path of the resulting CrySL specifications (Line 3), and the sets of abstract CrySL rules and refinements that should

```
1 config android25plus {
2   src = MetaCrySL/samples/jca/base/;
3   out = MetaCrySL/samples/jca/android/target/research/25plus/;
4   load spec base/;
5   load refinement android-bsi/01plus/;
6   load refinement android-bsi/10plus/;
7   load refinement android-bsi/1025/;
8 }
```

Figure 14 Example of a configuration that specifies the rules and refinements target to the version 25 of Android.

be considered during the building process (Lines 4–7). In the example, all CrySL rules reside in the `base` directory. One may also specify individual rules instead of a directory. The specification of a building process allows cryptographic experts to reuse the same specifications and refinements in different configurations. That is, from the same set of Lightweight Bouncy Castle specifications and refinements, we can create different configurations and generate distinct sets of CrySL rules. For this flexibility, we opted for such a *configuration language* instead of a *convention-based* mechanism.

5 Empirical Assessment of Meta-CrySL

The **goal** of this empirical assessment is to understand the implications of Meta-CrySL in modularizing the specifications of the correct usage of the JCA API for Android, and thereby evaluating Meta-CrySL along the lines of *compactness*. Additionally, we also use the empirical assessment to investigate whether or not Meta-CrySL generates correct CrySL specifications, focusing on the *correctness* dimension. Accordingly, we answer the following research **questions** in this empirical assessment, where RQ4 and RQ5 relate to *compactness* and RQ6 explore the *correctness* perspective:

RQ4 How many lines of CrySL code can one save when writing Meta-CrySL specifications?

RQ5 How much duplication of specifications is eliminated by using Meta-CrySL in comparison to CrySL?

RQ6 What are the implications of instantiating CrySL rules from Meta-CrySL specifications, observing the number of API misuses CogniCrypt$_{SAST}$ analysis reports?

Answering **RQ4** and **RQ5** allows us to quantify the main expected benefit of Meta-CrySL: modularizing families of CrySL specifications with the aim of specification reuse. Answering **RQ6** allows us (a) to contrast the difference in the number of reported API misuses when evaluating programs using different Meta-CrySL configurations and (b) to check the correctness of our approach for generating CrySL rules (since CogniCrypt$_{SAST}$ will reject any invalid CrySL rule). In this assessment, we used Meta-CrySL to modularize a family of CrySL specifications supporting different **versions** of the Android platform and three **sets of cryptographic recommendations**:

- **Android Base recommendations**: constrains the algorithms that should be used for each version of the Android platform, as detailed in the Android Cryptography Guide specification.[7]

[7] Android Cryptography Guide: `https://developer.android.com/guide/topics/security/cryptography`

Table 2 Sets of cryptographic rules considered in our study.

Config. Id	Primitives	Android Platform Version	Crypto Standard
C01	All primitives	01 – 08	Android Base recommendations
C02	All primitives	01 – 16	Android Base recommendations
C03	All primitives	01 – 28	Android Base recommendations
C04	All primitives	01 – 08	Android BSI Standard recommendations
C05	All primitives	01 – 16	Android BSI Standard recommendations
C06	All primitives	01 – 28	Android BSI Standard recommendations
C07	All primitives	01 – 08	Android CogniCrypt recommendations
C08	All primitives	01 – 16	Android CogniCrypt recommendations
C09	All primitives	01 – 28	Android CogniCrypt recommendations

- **Android BSI standard recommendations**: constrains the algorithms considering the BSI standard and the set of Android Base recommendations. The set of Android Base recommendations must be considered because not all BSI recommended algorithms are available in every version of the Android platform.
- **Android CogniCrypt recommendations**: constrains the algorithms according to the current CrySL specifications from the CogniCrypt project and the set of Android Base recommendations. The set of Android Base recommendations must be considered because not all CogniCrypt recommended algorithms are available in every version of the Android platform.

Specifying the correct usage of the JCA for Android is an interesting scenario for using Meta-CrySL, in particular because the correct usage of cryptography in Android depends on the version of the Android platform. Moreover, to answer our research question RQ6, this decision allows us to leverage the same dataset of Android applications that was previously used to empirically assess CrySL [24]. This dataset comprises 8,136 Android applications, though we could not collect the output of the CogniCrypt$_{SAST}$ for at least one configuration in a subset comprising 507 of these Android apps. For this reason, we consider a smaller set of 7,629 Android apps. From our Meta-CrySL specifications, we can generate hundreds of configurations. Since it is computationally expensive to run CogniCrypt$_{SAST}$ on a dataset with thousands of Android apps, we decided to conduct our assessment with the nine configurations shown in Table 2. Each configuration supports all cryptographic primitives (JCA supports 32 primitives in total, including Block Cipher and Message Digest), one of three distinct ranges of versions of the Android platform (01 – 08, 01 – 16, 01 – 28), and one of the cryptographic recommendations.

We answer research questions **RQ4**, **RQ5**, and **RQ6** through the use of metrics. For **RQ4** we compute (a) the total number of lines in Meta-CrySL necessary to specify the sets of configurations of Table 2 and (b) the resulting lines of specifications in CrySL that we generate using the Meta-CrySL specifications. We then compute how many lines of specification text we save using Meta-CrySL. For **RQ5** we estimate the total number of duplication in the Meta-CrySL specifications, as well as in the generated CrySL rules. We answer **RQ6** using the *total number of rule violations* that CogniCrypt$_{SAST}$ finds in the dataset of Android applications when using each distinct set of CrySL rule configurations.

5.1 RQ4: How many lines of CrySL code can one save when writing Meta-CrySL specifications?

In **RQ4**, we investigate the benefits of using Meta-CrySL w.r.t. removing the redundant code that one would write when specifying the sets of CrySL rules describing the correct usage of cryptographic APIs – tailored to the nine configurations in Table 2. Figure 15

summarizes the total number of lines needed to write the Meta-CrySL specifications, refinements, and configurations as well as the total number of lines of specifications generated by Meta-CrySL and that could be used to execute CogniCrypt$_{SAST}$ with the distinct configurations. In this case study, we wrote 1,407 lines in Meta-CrySL (762 lines of Meta-CrySL specifications, 540 lines of Meta-CrySL refinements, and 105 lines of Meta-CrySL configurations), and generated 7,105 lines of CrySL rules for those configurations, saving 80% of lines.

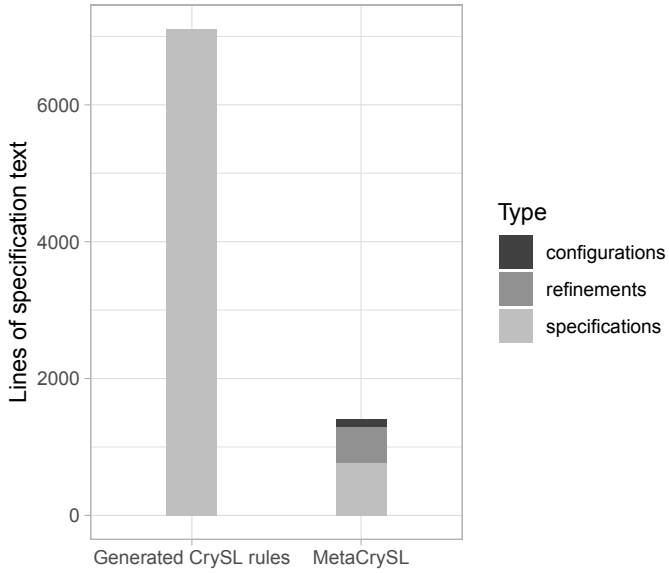


Figure 15 The total number of lines of code necessary to specify the nine configurations in Meta-CrySL (including Meta-CrySL specifications, Meta-CrySL refinements, and Meta-CrySL configurations) and the total number of lines of CrySL code generated.

Meta-CrySL removed 80% of the redundancy induced when writing all the CrySL rules tailored to the specific configurations considered in our study.

The Meta-CrySL payoff tends to increase when defining new configurations, since one would then generate further instances of CrySL from the same set of Meta-CrySL rules and refinements. Figure 16 shows how many lines of CrySL specification we generate after introducing each configuration in Table 2. In terms of lines of specification text, we achieve a payoff after generating the second configuration (C02).

5.2 RQ5: How much duplication of specifications is eliminated by using Meta-CrySL in comparison to CrySL?

In total, there were 188 files (including refinements and configurations) of base Meta-CrySL specifications for the JCA use in Android. These files contained 1407 lines of specifications, out of which 633 lines were duplicates, resulting out of 156 individual lines. In comparison, the corresponding CrySL specifications for three families of Android configurations (BSI, CogniCrypt, Base) each comprising specifications for three versions (0108, 0116, 25plus) contributed to 7,105 lines of specifications spread across 288 files. Out of these, 5,579 lines of specifications were duplicates resulting out of 546 unique lines.

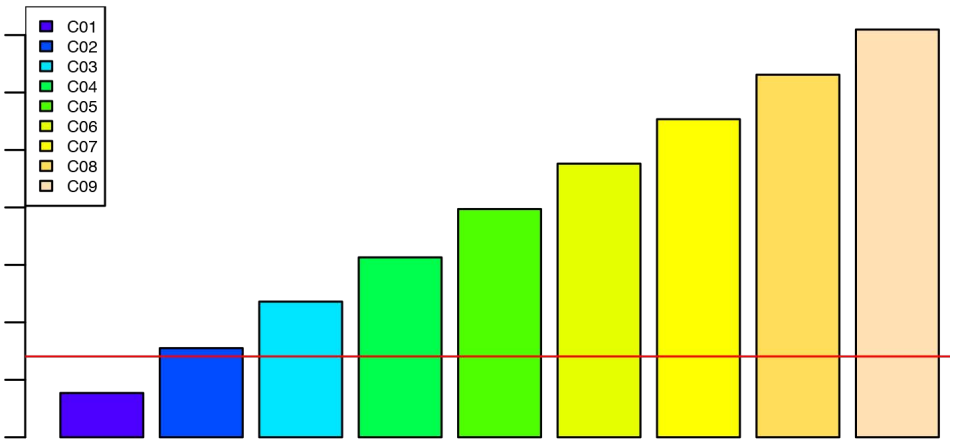


Figure 16 Evolution of the total lines of generated CrySL specification text after introducing each configuration. The red line corresponds to the total number of lines of Meta-CrySL used to generate the configurations.

> The amount of duplicate lines of specifications for a family of CrySL specifications is 5,579 in comparison to 633 for Meta-CrySL specifications for the same family (11.34%).

Most of the duplication in Meta-CrySL arises because we specified all Meta-CrySL refinements for the three families of Android configurations (BSI, CogniCrypt, and Base) which could be prevented by writing carefully crafted refinements. Specifically, out of the 1407 lines of specifications, only 97 lines and 85 lines of duplicates resulted from the base Meta-CrySL specifications and configurations – 451 of the 633 duplicates resulted from refinements for individual versions.

5.3 RQ6: What are the implications of instantiating CrySL rules from Meta-CrySL specifications, observing the number of API misuses CogniCrypt$_{sast}$ analysis reports?

Our research question RQ6 explores the results of CogniCrypt$_{SAST}$ for the nine configurations (C01 – C09). We concentrate our analysis on the violations related to the `CONSTRAINTS` section of CrySL rules, mostly because cryptographic standards do not address other sections. Table 3 summarizes the results of the analysis, showing the number of Android apps using the JCA APIs, the number of Android apps using the JCA APIs incorrectly (i.e., presenting at least one misuse), the rate of vulnerable Android apps (calculated using the previous two), and the total number of violations. The results of CogniCrypt$_{SAST}$ reveal a significant number of apps with at least one JCA API misuse in all configurations – more than five percent of the apps present at least one misuse in the more permissive Android Base sets of recommendations. This number jumps to more than 45% when considering the BSI or the CogniCrypt recommendations.

The total number of violations when considering the set of Android Base recommendations is substantially smaller than the total number of violation found using the other configurations (Android-BSI and Android-CC configurations) and most of the violations in the Android Base configurations relate to the *Cipher* primitive. For instance, when one only considers

Table 3 Summary of the findings of CogniCrypt$_{SAST}$ for the different CrySL configurations.

Configuration	Apps Using JCA	Apps Presenting Misuse	(Rate %)	# Violations
Android Base 0108	6,714	545	8.12	1,083
Android Base 0116	6,714	395	5.88	1,224
Android Base 25plus	6,714	386	5.75	830
Android BSI 0108	6,714	3,184	47.42	9,089
Android BSI 0116	6,714	3,155	46.99	8,905
Android BSI 25plus	6,714	3,155	46.99	8,873
Android CogniCrypt 0108	6,714	3,261	48.57	9,077
Android CogniCrypt 0116	6,714	3,260	48.56	8,975
Android CogniCrypt 25plus	6,714	3,256	48.50	8,945

the "Android Base 25plus" configuration, 548 out of 830 violations are either due to the use of an insecure cipher algorithm (such as DES or DESede) or due to the use of an insecure algorithm/mode/padding configuration (e.g., AES/CBC/NoPadding). This changes when one considers the other sets of recommendations (from Android BSI and Android CogniCrypt). There, most of the violations relate to the *Message Digest* primitive. For instance, considering the Android BSI `0116` configuration, one finds 6,272 violations due to insecure message-digest algorithms (e.g., MD5 and SHA-1) – this corresponds to 70.43% of all violations one finds with this particular configuration.

Regarding the differences between the Android BSI and Android CogniCrypt families of CrySL rules we found some modes of operations that are not mentioned in the BSI standard (e.g., RSA/ECB/PKCS1Padding) but that are considered secure and recommended in CogniCrypt. The *Message Authentication Code* (MAC) primitive also brings differences in the number of violations when comparing the BSI and CogniCrypt recommendations. Actually, the BSI standard makes clear that the HMAC scheme should only be used with the SHA-2 or SHA-3 families of hash functions, though the algorithms `HmacMD5` and `HmacSHA1` are allowed by the CogniCrypt configurations.

We also found some differences when considering the particular platform versions. For instance, until version 10 of the Android platform, developers must use the `TLS`[8] algorithm for `SSLContext`. This led to 169 additional violations regarding the incorrect usage of the `SSLContext` class in the "Android Base 0108" configuration, in comparison to "Android Base 0116" and "Android Base 25plus". In more detail, 161 apps use either the `SSL` or `TLSv1` algorithms (both introduced in version `10`) and eight apps use either `TLSv1.1` or `TLSv1.2` (both introduced in version `10`). These violations do not occur in the remaining "0116" and "25plus" configurations. We also found similar divergences on the platform version related to other cryptographic primitives.

It is important to note that, although version 8 was released in May 2010 already, in order to increase compatibility with a broader range of devices, most apps in our dataset are still configured to use this version as the minimum version. The observation that the number of violations for the "Android Base 0108" configuration is higher compared to the the "Android Base 0116" and "25plus" configurations might indicate that some apps use cryptographic algorithms that are not available in their minimum version. This would then lead to a runtime exception. In summary:

[8] TLS is a protocol that provides authenticated encryption for data connections.

The experiments showed a significant difference when considering the different versions of the platform for the Android Base configurations. Yet, the Android Base configurations are much less restrictive then those of the BSI and by CogniCrypt in general. We found slight differences in the results of CogniCrypt$_{SAST}$ when considering the recommendations from BSI and CogniCrypt. Although the differences are not that large, this result still suggests that one can benefit from tailoring the specifications of the correct usage of cryptographic standards according to the different guidelines.

6 Threats to Validity

In this section, we present some limitations and possible threats to the validity of our work. Since our research focuses on cryptographic libraries only, we need to discuss the applicability of our approach to other domains. The choice of this domain was motivated by our previous experience using CrySL to specify the correct usage of cryptographic APIs. We was challenged by the fact that new algorithms are frequently designed and old ones might become deprecated [5]. In addition, cryptographic standards are frequently updated – in particular to state that an algorithm vulnerability has been found and reported.

We believe that our approach can also be used for APIs that target other domains as well, even though we did not systematically investigate this question. First, APIs from different domains evolve along the time, and as we discussed throughout this paper, API evolution has an impact on the correct usage of libraries. Second, there are different recommendations on the proper usage of each popular API. For instance, there are many guidelines discussing the correct usage of the Java Persistence API [26, 30] – and individual companies might also take advantage of specific recommendations. The specifications about how to correctly use a given API should take into account these differences. We envision that both practitioners and researchers would benefit from a domain engineering approach that considers different sources of variability – including different versions of an API, recommendations from gray literature (for instance), and mining software repositories efforts – before specifying the correct usage a given API. We make the reader aware that domain engineering is a well-known technique to understand properties that, like in our case, bring variability to the domain of API usage specifications. We are not attempting to validate domain engineering itself or propose a technique for its application to other domains; the process for which would require careful understanding of the specific API domain and a thorough analysis.

Another threat to our conclusions relates to the additional complexity introduced by Meta-CrySL. We envision that the users of Meta-CrySL are already users of CrySL, and the learning curve would involve a language for specifying CrySL refinements and configurations. To better quantify the additional complexity Meta-CrySL introduces, we will have to conduct a *user study* with this specific goal. We postpone such an investigation to a future work, since our focus here was to explore Meta-CrySL in a more realistic scenario, investigating the possible benefits of using Meta-CrySL to modularize CrySL specifications for different versions of the Android platform and different cryptographic recommendations. Therefore, currently we do not have empirical evidence about how much complexity Meta-CrySL introduces to those already familiar with CrySL. Nonetheless, compared to the benefit of managing a large family of specifications using a relatively small number of refinements and configurations, we feel this additional complexity is justified.

Additional threats relate to the methods we used in our research. We tried to mitigate possible *reliability threats* by reusing methods and tools from previous research studies. For instance, we investigated the frequency of *breaking changes* to estimate the stability

of Java cryptographic libraries using the methodologies available in the literature [7, 8, 49]. Nonetheless, although we found more than 1700 *breaking changes* across 11 public releases of Bouncy Castle, a limitation of our work is that we do not investigate if these changes have actually broken existing client code. Our understanding is that just a subset of breaking changes impact on the specifications of the correct usage of APIs.

This threat relates to the use of APIDIFF, which detects breaking changes considering modifications to the *standard notion* of Java interfaces – that is, public members of Java classes or interfaces. Modifications that do not preserve the standard notion of Java interfaces (e.g., changing the signature of public methods, removing public methods, and so on) are claimed by APIDIFF as breaking changes. This might actually lead to a number of false-positives – once client code might not depend on all public members of a library. To mitigate this threat, we narrowed our analysis of the Bouncy Castle library to only focus on the high-level classes and interfaces of Bouncy Castle that implement cryptographic primitives.

Regarding our research question **RQ4**, we measure the reduction of lines of specification and redundancy with respect to *generated* specifications. This might raise the question whether these generated specifications do not contain boilerplate text that had not arisen had these specifications be hand-written. We are confident that we can rule this out, due to the nature of CrySL specifications and the way they are generated by Meta-CrySL. Conducting a large scale developer study by manually writing many families of specifications by hand was beyond the scope of this work.

7 Related Work

7.1 Domain Engineering

Frakes et al. [17] present a well-established definition for domain engineering, which embraces two phases: *domain analysis* and *domain implementation*. The first deals with all activities necessary to understand and document the commonalities and variabilities within a software domain. Similar to the guidelines presented by the authors, we also collected and recorded information from documents (cryptographic standards) and source code (examples of cryprographic libraries usage) while conducting our domain analysis. The main difference of our approach is that we also mined the source code evolution of the cryptographic libraries. Lisboa et al. presents a literature review on tools and methods for domain analysis [28].

The second phase of domain engineering (that is, domain implementation) aims to build the infrastructure necessary to generate products from reusable assets. Here we used the same general idea, though not to build software products, but actually to generate specifications of the correct usage of APIs that might vary according to different sources of variablity (such as versions of APIs, platforms, and cryptographic standards). Czarnecki and Eisenecker [10] detail several techniques that can be used to implement an infrastructure for building products from reusable assets. In our work, we used the *refinement-based* transformational approach [10, Chapter 9] as the basis for the Meta-CrySL design and implementation. The literature on software product lines also recommends two distinct phases for building SPLs: one for domain analysis and one for domain implementation [4, 36].

7.2 Correct Usage of APIs

Amann et al. present some terminology and taxonomy around the correct usage and misuse of APIs [3]. Given a set of constraints stating, for instance, the expected order of method calls and the pre-conditions the client code must guarantee before calling the methods of an

API, any usage scenario that violates a constraint characterizes a misuse – otherwise, it is a correct usage [3]. The main goal of mining misuse of APIs is to reveal *deviant code* that might originate a bug or a software vulnerability (in the context of cryptographic APIs).

According to Amann et al [3], the constraint specifications could be manually crafted by experts or infered using either dynamic [29, 37] or static analysis [32, 40, 48]. In this paper, we rely on a manually crafted approach to specify rules in Meta-CrySL– mostly because many programs fail to use cryptographic APIs correctly [1, 24, 33]. It is a matter of future work to investigate if our domain engineering approach could also benefit from techniques that automatically infer the correct usage of APIs.

To the best of our knowledge, none of the previous research works consider that the correct usage of an API could vary, among other reasons, according to specific versions of APIs or to existing usage recommendation patterns that could be general accepted or tailored to particular companies or projects.

7.3 API Evolution

Studies on API evolution focus on two directions. First, to help developers to migrate their systems in response to the evolution of APIs the systems depend on [9, 18, 31, 43]. The second direction, which is closely related to our research, focus on understanding how developers evolve APIs and on characterizing the evolution of APIs. For instance, several research works have explored the impact of *deprecation* mechanisms on software ecosystems [39, 41, 42]. Other research studies investigate how developers respond to API evolution [19] and the motivations for breaking APIs [6].

Here we investigate how the evolution of cryptographic APIs occurs in practice, considering the history of three Java cryptographic libraries: JCA/JCE, Bouncy Castle, and Google Tink. We have found that cryptographic libraries are quite stable, and the high-level APIs that define cryptographic primitives rarely change – even though we found a number of *breaking changes* during the evolution of Bouncy Castle. The most typical pattern is the introduction of new algorithms that implement cryptographic primitives – which often requires changes into the specification about the correct usage of the APIs.

8 Conclusion

Domain engineering involves a set of techniques for identifying and documenting the commonalities and variabilities within a software domain, as well as for building an infrastructure for deriving products from reusable assets [4, 17, 36]. While it has been successfully used to develop software product lines, in this paper, we explored the use of domain engineering procedures to specify the correct usage of cryptographic APIs. After gathering a better understanding about how different versions of the platforms, APIs, and cryptographic standards might affect the specifications of the correct usages of crypto APIs, we designed Meta-CrySL. Meta-CrySL serves as an infrastructure for generating CrySL [24] specifications tailored for specific scenarios. We evaluated our approach using a family of Meta-CrySL specifications describing the correct usage of the Java Cryptographic Architecture for Android, which accommodates the evolution of the Android platform and three distinct sets of cryptographic recommendations. Our results provide evidence that it is important to tackle the problem of writing specifications of correct usage of APIs using a domain engineering approach and that using Meta-CrySL we can better modularize families of specifications.

References

1 Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. Comparing the usability of cryptographic apis. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 154–171. IEEE Press, May 2017. doi:10.1109/SP.2017.52.

2 S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering*, 45(12):1170–1188, 2019.

3 S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering*, 45(12):1170–1188, 2019. doi:10.1109/TSE.2018.2827384.

4 Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013.

5 A. Bhardwaj and S. Som. Study of different cryptographic technique and challenges in future. In *2016 International Conference on Innovation and Challenges in Cyber Security (ICICCS-INBUSH)*, pages 208–212, 2016.

6 Aline Brito, Marco Tulio Valente, Laerte Xavier, and André C. Hora. You broke my code: understanding the motivations for breaking changes in apis. *Empirical Software Engineering*, 25(2):1458–1492, 2020. doi:10.1007/s10664-019-09756-z.

7 Aline Brito, Laerte Xavier, André C. Hora, and Marco Tulio Valente. Apidiff: Detecting API breaking changes. In Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd, editors, *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 507–511. IEEE Computer Society, 2018. doi:10.1109/SANER.2018.8330249.

8 Aline Brito, Laerte Xavier, André C. Hora, and Marco Tulio Valente. Why and how Java developers break APIs. In Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd, editors, *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 255–265. IEEE Computer Society, 2018. doi:10.1109/SANER.2018.8330214.

9 Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *1996 International Conference on Software Maintenance (ICSM '96), 4-8 November 1996, Monterey, CA, USA, Proceedings*, page 359. IEEE Computer Society, 1996. doi:10.1109/ICSM.1996.565039.

10 Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., USA, 2000.

11 Danny Dig and Ralph Johnson. How do apis evolve? a story of refactoring: Research articles. *J. Softw. Maint. Evol.*, 18(2):83–107, March 2006.

12 Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 73–84, New York, NY, USA, 2013. ACM. doi:10.1145/2508859.2516693.

13 Michel Abdalla et al. Algorithms, key size and protocols report. Technical report, ECRYPT – Coordination and Support Action, European Union's H2020 programme, 2018.

14 Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010.

15 F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack overflow considered harmful? the impact of copy amp;paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136, May 2017. doi:10.1109/SP.2017.31.

16 German Federal Office for Information Security. Cryptographic mechanisms: Recommendations and key lengths. Technical Report BSI TR-02102-1, German Federal Office for Information Security, 2020.

17 William Frakes, Ruben Prieto, Christopher Fox, et al. Dare: Domain analysis and reuse environment. *Annals of software engineering*, 5(1):125–141, 1998.

18 Johannes Henkel and Amer Diwan. Catchup! capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, page 274–283, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1062455.1062512.

19 A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. Tulio Valente. How do developers react to api evolution? the pharo ecosystem case. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 251–260, 2015. doi:10.1109/ICSM.2015.7332471.

20 David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004. doi:10.1145/1052883.1052895.

21 Oracle Inc. Java cryptography architecture (JCA), 2020. URL: https://docs.oracle.com/en/java/javase/15/security/java-cryptography-architecture-jca-reference-guide.html.

22 Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. Effective and efficient api misuse detection via exception propagation and search-based testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 192–203, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293882.3330552.

23 Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, pages 168–177. IEEE Computer Society, 2009. doi:10.1109/SCAM.2009.28.

24 S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, pages 1–1, 2019. doi:10.1109/TSE.2019.2948910.

25 Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. Cognicrypt: Supporting developers in using cryptography. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 931–936. IEEE Press, 2017.

26 A. Leonard. *Spring Boot Persistence Best Practices: Optimize Java Persistence Performance in Spring Boot Applications*. Apress, 2020. URL: https://books.google.com.br/books?id=dIvgDwAAQBAJ.

27 Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. icryptotracer: Dynamic analysis on misuse of cryptography functions in ios applications. In Man Ho Au, Barbara Carminati, and C.-C. Jay Kuo, editors, *Network and System Security*, pages 349–362, Cham, 2014. Springer International Publishing.

28 Liana Barachisio Lisboa, Vinicius Cardoso Garcia, Daniel Lucrédio, Eduardo Santana de Almeida, Silvio Romero de Lemos Meira, and Renata Pontin de Mattos Fortes. A systematic review of domain analysis tools. *Information and Software Technology*, 52(1):1–13, 2010. doi:10.1016/j.infsof.2009.05.001.

29 Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin ŞerbănuŢă, and Grigore Roşu. Rv-monitor: Efficient parametric runtime verification with simultaneous properties. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 285–300, Cham, 2014. Springer International Publishing.

30 Dustin Marx. Basic java persistence api best practices. Technical report, Oracle, 2008.

31 Mira Mezini. Maintaining the consistency of class libraries during their evolution. *SIGPLAN Not.*, 32(10):1–21, 1997. doi:10.1145/263700.263701.

32 Martin Monperrus and Mira Mezini. Detecting missing method calls as violations of the majority rule. *ACM Trans. Softw. Eng. Methodol.*, 22(1), 2013. doi:10.1145/2430536.2430541.

33 Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 935–946. ACM, 2016. doi:10.1145/2884781.2884790.

34 National Institute of Standards and Technology. Security requirements for cryptographic modules. Technical report, National Institute of Standards and Technology, 2019.

35 Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 1st edition, 2009.

36 Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin, Heidelberg, 2005.

37 Michael Pradel and Thomas R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 288–298. IEEE Press, 2012.

38 Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2455–2472, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3319535.3345659.

39 Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to api deprecation? the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2393596.2393662.

40 M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. Mining multi-level api usage patterns. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 23–32, 2015. doi:10.1109/SANER.2015.7081812.

41 A. A. Sawant, R. Robbes, and A. Bacchelli. On the reaction to deprecation of 25,357 clients of 4+1 popular java apis. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 400–410, 2016. doi:10.1109/ICSME.2016.64.

42 Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of clients of 4+ 1 popular java apis and the jdk. *Empirical Software Engineering*, 23(4):2158–2197, 2018.

43 Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 471–480, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1368088.1368153.

44 Bruce Schneier. *Secrets & Lies: Digital Security in a Networked World*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 2000.

45 Johannes Späth, Karim Ali, and Eric Bodden. Ide^{al}: efficient and precise alias-aware dataflow analysis. *PACMPL*, 1(OOPSLA):99:1–99:27, 2017. doi:10.1145/3133923.

46 Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *PACMPL*, 3(POPL):48:1–48:29, 2019.

47 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 22:1–22:26. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.22.

48 Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 35–44, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1287624.1287632.

49 L. Xavier, A. Brito, A. Hora, and M. T. Valente. Historical and impact analysis of api breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147, February 2017. doi:10.1109/SANER.2017.7884616.

On the Monitorability of Session Types, in Theory and Practice

Christian Bartolo Burlò
Gran Sasso Science Institute, L'Aquila, Italy

Adrian Francalanza
Department of Computer Science, University of Malta, Msida, Malta

Alceste Scalas
DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark

Abstract

Software components are expected to communicate according to predetermined protocols and APIs. Numerous methods have been proposed to check the correctness of communicating systems against such protocols/APIs. *Session types* are one such method, used both for static type-checking as well as for run-time monitoring. This work takes a fresh look at the run-time verification of communicating systems using session types, in theory and in practice. On the theoretical side, we develop a formal model of session-monitored processes. We then use this model to formulate and prove new results on the *monitorability* of session types, defined in terms of *soundness* (*i.e.,* whether monitors only flag ill-typed processes) and *completeness* (*i.e.,* whether all ill-typed processes can be flagged by a monitor). On the practical side, we show that our monitoring theory is indeed *realisable*: we instantiate our formal model as a Scala toolkit (called `STMonitor`) for the automatic generation of session monitors. These executable monitors can be used as proxies to instrument communication across black-box processes written in any programming language. Finally, we evaluate the viability of our approach through a series of benchmarks.

2012 ACM Subject Classification Software and its engineering → Development frameworks and environments; Software and its engineering → Software verification and validation; Theory of computation → Concurrency

Keywords and phrases Session types, monitorability, monitor correctness, Scala

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.20

Related Version *Full Version*: `https://arxiv.org/abs/2105.06291`

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.2`

Funding This work has been partly supported by: the project MoVeMnt (No: 217987-051) under the Icelandic Research Fund; the BehAPI project funded by the EU H2020 RISE under the Marie Skłodowska-Curie action (No: 778233); the EU Horizon 2020 project 830929 *CyberSec4Europe*; the Danish Industriens Fonds Cyberprogram 2020-0489 *Security-by-Design in Digital Denmark*.

1 Introduction

Communication protocols and Application Programming Interfaces (APIs) [18] govern the interactions between concurrent and distributed software components by exposing the functionality of a component for others to use. Although the order of messages exchanged and methods invoked is crucial for correct API usage, this information is either outright omitted, or stated informally via textual descriptions [62, 61]. At best, protocols and temporal API usage are described semi-formally as message sequence charts [51]. This state of affairs is conducive to conflicting interactions, which may manifest themselves as run-time errors, deadlocks and livelocks. *Behavioural types* [11] provide a methodology

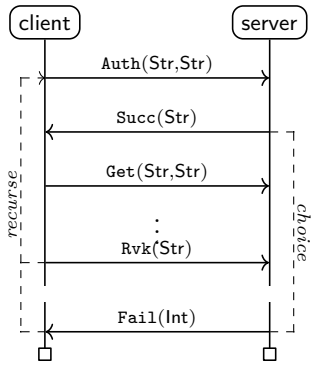


Figure 1 Authentication Protocol.

to address these shortcomings, by elevating protocols and flat API descriptions to *formal behavioural specifications* with explicit *sequences* and *choices* of operations. A prevalent form of behavioural types are *session types* [36, 37] which can ensure correct interactions that are free from communication errors, deadlocks and livelocks.

▶ **Example 1.** Consider a server that exposes the API calls `Auth` (authenticate), `Get` and `Rvk` (revoke). The intended use of this API is to invoke `Auth` followed with `Get` and finally `Rvk`, as depicted in Fig. 1. If authentication is successful, `Auth` returns a token that can be used for exclusive access to a resource with the service `Get`. After its use, the token should be revoked with the service `Rvk` to allow other parties to access the resource. For security reasons, the server is expected to only reply `Get` requests after it services an `Auth` request. However, if the order of invocation is not respected, a client may send a `Get` request before an `Auth` request. The resulting components' interaction will be incorrect, causing an error or deadlock. Even worse, the server may accept the `Get` request and let an unauthenticated client access sensitive information. The protocol from the viewpoint of the client can be described as the session type:

$$S \quad = \quad !\texttt{Auth}\,.\,\&\left\{\ ?\texttt{Succ}\,.!\texttt{Get}\ldots!\texttt{Rvk}\,.\,S,\quad ?\texttt{Fail}\,.\,S\ \right\}$$

Type S states that the client is expected to first invoke (!) the service `Auth` and then branch (&) according to the response received (?). If it receives `Success`, the client can invoke `Get` and eventually `Rvk` before restarting the protocol (S). Otherwise, if it receives `Fail`, the client may start following the type S from the beginning and retry authentication. ⌟

Run-time monitoring of session types: promise and challenges. In behavioural type frameworks (including session types), the conformance between the component under scrutiny and a desired protocol is commonly checked *statically*, via a type system that is tailored for the language used to develop the component. This avoids runtime overhead and allows for early error detection. However, there are cases where a (full) static analysis is not possible. For instance, within a distributed or collaborative system, not all system components are necessarily accessible for static analysis (*e.g.,* due to source obfuscation). Components may also be implemented using different programming languages, making it infeasible to develop bespoke type-checkers for every programming language used in development. In these cases, post-deployment techniques such as Runtime Verification (RV) [29, 13] can be used where protocol conformance is carried out *dynamically* via *monitors* [21, 50, 24, 42, 49, 17, 34]. Runtime monitoring of behavioural types comes with a set of challenges.

The realisability of effective monitoring: Restrictions such as inaccessible code and license agreements (regulating code modifications), may restrict the ways in which software components can be instrumented, thus hindering a monitor's capabilities for observation and intervention. Moreover, the runtime overhead induced by monitors should be kept within acceptable levels.

Monitor Correctness: Intuitively, a "correct" monitor for a session type S should carry out detections that correspond to the protocol represented by S. The recent results on *monitorability* help us unpack this intuition of "correctness" in terms of *soundness* and *completeness*: the monitor should not unnecessarily flag well-behaving code (*detection soundness* [30, 3]), while providing guarantees for recognising misbehaving components (*detection completeness* [4, 6]).

The aforementioned challenges are not independent of one another, and an adequate solution often needs to take both aspects into consideration. On the one hand, monitor correctness may require computations that increase runtime overheads; on the other hand, there are inherent limits to what can be detected at runtime (*i.e.,* the monitorability problem [13]) – and moreover, practical implementation concerns may restrict monitoring capabilities even further (*e.g.,* due to the need for low overheads). To our knowledge, the above aspects have not been fully investigated together for session types, in *one unified study*:

- there is no systematic examination for the *monitorability* of session types, determining the limits of runtime monitoring when verifying session-type conformance;
- no previous work tackles the design of a session monitoring system that is practically *realisable*, while also backed by formal detection soundness *and* completeness guarantees.

Contributions. We present the first formal analysis of the *monitorability of session types*, and use it to guide the design and implementation of a practical framework (written in Scala) for the run-time monitoring of concurrent and distributed applications. We focus on communication protocols that can be formalised as *(binary) session types* [36, 37] with two interacting parties (*e.g.,* a client and a server). Crucially, we tackle scenarios where at least one of the parties is a "black-box" process that may not be statically verified. After formalising a streamlined process calculus with session types (§ 2), we present our contributions:

1. We develop a formal model detailing how processes can be instrumented with monitors, to observe their interactions and flag violations on the offending party (§ 3). We then design an automated synthesis procedure from session types to monitors (in this operational model) to study the monitorability of session types (§ 3.4);
2. We carry out the first study on the *monitorability* of session types, formally linking their static and run-time verification (§ 4). We prove that our synthesised monitors are *detection-sound*, *i.e.,* components flagged by a monitor for session type S are indeed ill-typed for S (Theorem 15). We also prove a *weak detection-completeness* result (Theorem 19) showing to what degree can our synthesised monitors detect ill-typed components. Importantly, we show that these limits are not specific to our synthesis procedure by proving an impossibility result: under our "black-box" monitoring model, session monitoring cannot be both sound and complete (Theorem 21). The latter results are new to the area of behavioural types;
3. We show the *realisability* of our model, by implementing a toolkit (called `STMonitor`) that synthesises session monitors as executable Scala programs (§ 5). We provide `STMonitor` as companion artifact of this paper. We also provide evaluation benchmarks showing that our generated Scala monitors induce limited overheads, hence their usability in practice appears promising (§ 6).

   Proofs and additional details are available in the extended version of this paper [19].

Syntax

$$\begin{array}{lrcl} \text{Predicates} & A & ::= & \mathsf{tt} \mid \mathsf{ff} \mid v_1 == v_2 \mid v_1 >= v_2 \mid A_1 \;\&\&\; A_2 \mid !A \mid \ldots \\ \text{Processes} & P, Q & ::= & \triangleleft \mathtt{l}(a).P \mid \triangleright \left\{ \mathtt{l}_i(x_i).P_i \right\}_{i \in I} \mid \mu_X.P \mid X \mid \mathsf{if}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid \mathbf{0} \end{array}$$

Semantics

$$[\textsc{pRec}]\ \frac{}{\mu_X.P \xrightarrow{\tau} P[\mu_X.P/X]} \qquad [\textsc{pSnd}]\ \frac{}{\triangleleft \mathtt{l}(v).P \xrightarrow{\triangleleft \mathtt{l}(v)} P}$$

$$[\textsc{pRcv}]\ \frac{j \in I}{\triangleright \left\{ \mathtt{l}_i(x_i).P_i \right\}_{i \in I} \xrightarrow{\triangleright \mathtt{l}_j(v)} P_j[v/x_j]}$$

$$[\textsc{pTru}]\ \frac{A \Downarrow \mathsf{tt}}{\mathsf{if}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} P} \qquad [\textsc{pFls}]\ \frac{A \Downarrow \mathsf{ff}}{\mathsf{if}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} Q}$$

Figure 2 Process Calculus Syntax and Semantics.

2 Process Calculus and Session Types

This section introduces the formalism at the basis of our work: a streamlined process calculus (§ 2.1) with standard session types (§ 2.2) and typing system (§ 2.3).

2.1 Process Calculus

Syntax. We adopt a streamlined process calculus that models a sequential process interacting on a single communication channel, similar to [33, 32, 60]. Our process calculus is defined in Figure 2. The syntax assumes separate denumerable sets of **values** $v, u, w \in \textsc{Val}$ (including tuples), value **variables** $x, y, z \in \textsc{Var}$ and **process variables** $X, Y \in \textsc{PVar}$. We use a, b to range over the set $\textsc{Val} \cup \textsc{Var}$. The syntax also assumes a set of **predicates** A (used in conditionals). A process may communicate by sending or receiving **messages** of the form $\mathtt{l}(v)$, where $\mathtt{l}$ is a **label**, and v is the **payload value**. To this end, a process may perform **outputs** $\triangleleft \mathtt{l}(a).P$ (*i.e.,* send message $\mathtt{l}(v)$ and continue as P), or **inputs** $\triangleright \left\{ \mathtt{l}_i(x_i).P_i \right\}_{i \in I}$ (*i.e.,* receive a message with label $\mathtt{l}_i$ for any $i \in I$, and continue as P_i, with x_i replaced by the message payload). Loops are supported by the **recursion** construct $\mu_X.P$, and the **process variable** X. The process $\mathbf{0}$ represents a **terminated** process. The calculus also includes a standard **conditional** construct $\mathsf{if}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q$. We assume that all recursive processes are **guarded**, *i.e.,* process variables can only occur under an input or output prefix. The calculus has two **binders**: the input construct $\triangleright \left\{ \mathtt{l}_i(x_i).P_i \right\}_{i \in I}$ binds the free occurrences of the (value) variables x_i in the continuation process P_i, whereas the recursion construct $\mu_X.P$ binds the process variable X in the continuation process P.

Semantics. The dynamic behaviour of a process is described by the transition rules in Fig. 2. The rules take the form $P \xrightarrow{\mu} P'$, where the transition **action** μ can be either an **output action** $\triangleleft \mathtt{l}(v)$, an **input action** $\triangleright \mathtt{l}(v)$, or a **silent action** τ. Rule [pRec] allows the recursive process $\mu_X.P$ to unfold. Rules [pSnd] and [pRcv] enable communication:

- by [pSnd], process $\triangleleft \mathtt{l}(v).P$ sends a message by firing action $\triangleleft \mathtt{l}(v)$ and continuing as P;
- by [pRcv], process $\triangleright \left\{ \mathtt{l}_i(x_i).P_i \right\}_{i \in I}$ can receive a message $\mathtt{l}_j(v)$ $(j \in I)$ by firing action $\triangleright \mathtt{l}_j(v)$ and continuing as P_j, with the payload value v replacing the variable x_j.

The remaining two rules [pTru] and [pFls] define the silent transitions when the predicate in the process $\mathsf{if}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q$ evaluates to true ($A \Downarrow \mathsf{tt}$) or false ($A \Downarrow \mathsf{ff}$), respectively. For brevity, we often omit the trailing $\mathbf{0}$ and write $\triangleright \mathtt{l}(v).P$ for singleton input choices.

$$\textbf{Syntax}\quad \begin{array}{rrcl} & \text{Base types} & \mathsf{B} & ::= \ \mathsf{Int} \mid \mathsf{Str} \mid \mathsf{Bool} \mid \ldots \mid (\mathsf{B},\mathsf{B}) \\ & \text{Session types} & R,S & ::= \underbrace{\oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I} \mid \&\{?\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}}_{\text{with } I\neq\emptyset \text{ and } \mathtt{l}_i \text{ pairwise distinct}} \mid \mathsf{rec}\ X.S \mid X \mid \mathsf{end} \end{array}$$

$$\textbf{Dual types}\quad \begin{array}{rcl} \overline{\&\{?\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}} &=& \oplus\{!\mathtt{l}_i(\mathsf{B}_i).\overline{S_i}\}_{i\in I} \qquad \overline{\mathsf{end}} = \mathsf{end} \qquad \overline{X} = X \\ \overline{\oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}} &=& \&\{?\mathtt{l}_i(\mathsf{B}_i).\overline{S_i}\}_{i\in I} \qquad \overline{\mathsf{rec}\ X.S} = \mathsf{rec}\ X.\overline{S} \end{array}$$

Figure 3 Session Types Syntax, and Definition of Dual Types.

▶ **Example 2** (Process syntax and semantics)**.** Recall the protocol depicted in Fig. 1. A corresponding client process for this protocol is defined as P_{auth} below.

$$P_{auth} = \mu_X.\triangleleft\mathtt{Auth}(\mathtt{"Bob"},\mathtt{"pwd"}).P_{res} \qquad \text{where } P_{res} = \triangleright\{\mathtt{Succ}(tok).P_{succ}\,,\ \mathtt{Fail}(code).P_{fail}\}$$

From the rules in Figure 2, the process P_{auth} executes as follows:

$$\begin{array}{rl} P_{auth} \xrightarrow{\tau} \left(\triangleleft\,\mathtt{Auth}(\mathtt{"Bob"},\mathtt{"pwd"}).P_{res}\right)[P_{auth}/X] & \text{using [pRec]} \\ \xrightarrow{\triangleleft\mathtt{Auth}(\mathtt{"Bob"},\mathtt{"pwd"})} \triangleright\left\{\begin{array}{l}\mathtt{Succ}(tok).P_{succ}[P_{auth}/X]\,,\\ \mathtt{Fail}(code).P_{fail}[P_{auth}/X]\end{array}\right\} & \text{using [pSnd]} \end{array}$$

The process performs a silent action τ to unfold its recursion, and then sends a message with label `Auth` and tuple `"Bob","pwd"` as payload. If the authentication is successful, the process receives the message `Succ` including a token *tok* and proceeds according to P_{succ} (omitted):

$$\xrightarrow{\triangleright\mathtt{Succ}(321)} P_{succ}[P_{auth}/X][321/tok] \qquad \text{using [pRcv]}$$

Otherwise, if the authentication is unsuccessful, the process receives the message `Fail` including an error *code* from the server and proceeds according to P_{fail}. ⌟

2.2 Binary Session Types

Session types describe the structure of interaction among processes. They enable the verification of communicating systems against a stipulated communication protocol. Figure 3 formalises binary session types. We assume a set of standard base types B which includes tuples. The **selection type** (or **internal choice**) $\oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}$ requires a component to send a message $\mathtt{l}_i(v)$ where the value v has base type B_i, for some $i \in I$. The **branching type** (or **external choice**) $\&\{?\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}$ requires a component to receive a message of the form $\mathtt{l}_i(v)$, where the value v (*i.e.,* the message payload) is of the corresponding base type B_i for any $i \in I$. The **recursive** session type $\mathsf{rec}\ X.S$ binds the recursion variable X in S (we assume guarded recursion), while end describes a **terminated** session. For brevity, we often omit $\oplus$ and & for singleton choices, as well as trailing ends.

A process implementing a session type S can correctly interact with a process implementing the **dual type** of S, denoted as $\overline{S}$ (defined in Fig. 3). Intuitively, the dual type of a selection is a branching type with the same choices. Hence, every possible output from one component matches an input by the other component, and *vice versa*. Duality guarantees that the interaction between typed components is *safe* (*i.e.,* only expected messages are communicated) and *deadlock-free* (*i.e.,* the session terminates only if both components reach their end).

▶ **Example 3.** The session type S_{auth} below formalises the first part of the protocol that the *client* in Fig. 1 is expected to follow (*i.e.,* the type S in Example 1).

$$S_{auth} = \mathsf{rec}\ Y.!\mathtt{Auth}(\mathsf{Str},\mathsf{Str}).\&\{?\mathtt{Succ}(\mathsf{Str}).S_{succ}\,,\ ?\mathtt{Fail}(\mathsf{Int}).Y\}$$

Identifier Typing

$$[\textsc{tVar}]\frac{\Gamma(x) = \mathsf{B}}{\Gamma \vdash x : \mathsf{B}} \qquad [\textsc{tVal}]\frac{v \in \mathsf{B}}{\Gamma \vdash v : \mathsf{B}}$$

Process Typing

$$[\textsc{tBra}]\frac{\forall i \in I \qquad \Theta \cdot \Gamma, x_i : \mathsf{B}_i \vdash P_i : S_i}{\Theta \cdot \Gamma \vdash \triangleright\{\mathtt{l}_i(x_i).P_i\}_{i\in I\cup J} : \&\{?\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}} \qquad [\textsc{tRec}]\frac{\Theta, X : S \cdot \Gamma \vdash P : S}{\Theta \cdot \Gamma \vdash \mu_X.P : S}$$

$$[\textsc{tSel}]\frac{\exists i \in I \qquad \mathtt{l} = \mathtt{l}_i \qquad \Gamma \vdash a : \mathsf{B}_i \qquad \Theta \cdot \Gamma \vdash P : S_i}{\Theta \cdot \Gamma \vdash \triangleleft\mathtt{l}(a).P : \oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}} \qquad [\textsc{tPVar}]\frac{\Theta(X) = S}{\Theta \cdot \Gamma \vdash X : S}$$

$$[\textsc{tIf}]\frac{\Gamma \vdash A : \mathsf{Bool} \qquad \Theta \cdot \Gamma \vdash P : S \qquad \Theta \cdot \Gamma \vdash Q : S}{\Theta \cdot \Gamma \vdash \mathsf{if}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q : S} \qquad [\textsc{tNil}]\frac{}{\Theta \cdot \Gamma \vdash \mathbf{0} : \mathsf{end}}$$

Figure 4 Session Typing Rules.

The *server* should follow $\overline{S_{auth}} = \mathsf{rec}\ Y.?\mathtt{Auth}(\mathsf{Str},\mathsf{Str}).\oplus\{!\mathtt{Succ}(\mathsf{Str}).\overline{S_{succ}}, !\mathtt{Fail}(\mathsf{Int}).Y\}$, its dual. According to S_{auth}, the *client* initiates interaction by sending a message with label `Auth`, carrying a tuple of strings (username and password) as payload. The *server* should then reply with either `Success` (carrying a string), or `Failure` (with an integer error code). In case of `Success`, the *client* continues along S_{succ}. In case of `Failure`, the session loops. ⌟

2.3 Session Typing System

Our session typing system (in Fig. 4) is standard. It uses two **typing environments** Θ and Γ, where Θ is a partial mapping from process variables to session types, while Γ is a partial mapping from value variables to base types. We represent them syntactically as:

$$\Theta ::= \emptyset \mid \Theta, X : S \qquad \Gamma ::= \emptyset \mid \Gamma, x : \mathsf{B}$$

The type system is *equi-recursive* [53]: when comparing two types, we consider a recursive type $\mathsf{rec}\ X.S$ to be equivalent to its unfolding $S[^{\mathsf{rec}\ X.S}/_X]$ (*i.e.,* interchangeable in all contexts).

The typing judgement for values and variables is $\Gamma \vdash a : \mathsf{B}$, defined by rules [tVar] and [tVal]. The process typing judgement, $\Theta \cdot \Gamma \vdash P : S$, states that process P communicates according to session type S, given the typing assumptions in Θ and Γ. In the *branching rule* [tBra], an input process has a branching type $\&\{?\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}$ if all the possible branches in the type are present as choices in the process, with matching labels. Hence, the process must have the form $\triangleright\{\mathtt{l}_i(x_i).P_i\}_{i\in I\cup J}$ (notice that if $J \neq \emptyset$, the process has more input branches than the type). Moreover, for each matching branch, each continuation process P_i (for $i \in I$) must be typed with the corresponding continuation type S_i, assuming that the received message payload x_i has the expected type B_i. The *selection rule* [tSel] states that $\triangleleft\mathtt{l}(a).P$ follows a selection type of the form $\oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}$ if there exists a possible choice in the type that matches the message $\mathtt{l}(a)$. To match, the labels must be identical, and the type of the payload a must be of the type B_i stated in the session type, and the continuation process P must be of the continuation type S_i. The remaining rules are fairly standard.

▶ Remark 4. Although we do not fix the boolean predicates A, we assume that:

1. boolean predicates can be type-checked with standard rules;
2. base types B come with a predicate $\mathsf{isB}(v)$ that returns tt if v is of type B, and ff otherwise (akin to `instanceof` in Java.) ⌟

▶ **Example 5.** Recall the process P_{auth} (Example 2) and the session type S_{auth} (Example 3):

$$P_{auth} = \mu_X . \triangleleft \texttt{Auth}(\texttt{"Bob"}, \texttt{"pwd"}).P_{res} \qquad S_{auth} = \mathsf{rec}\ Y.!\texttt{Auth}(\mathsf{Str}, \mathsf{Str}).S_{res}$$

One can show that P_{auth} type-checks with S_{auth}, *i.e.*, $\emptyset \cdot \emptyset \vdash P_{auth} : S_{auth}$. ⌟

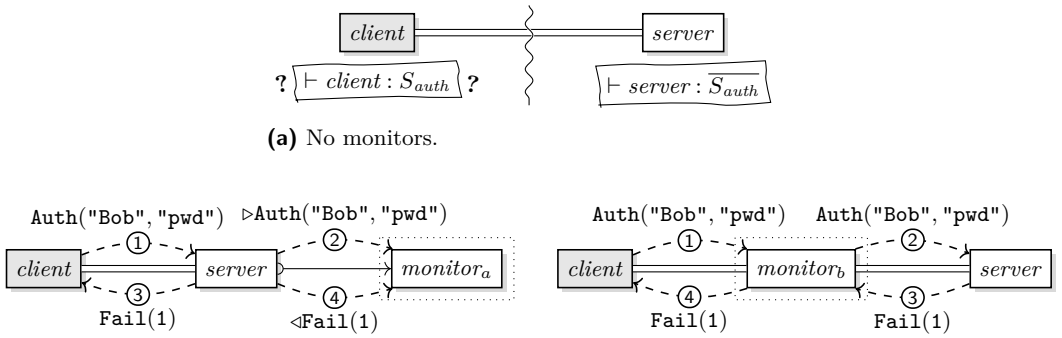
3 A Formal Model for Monitoring Sessions

We now formalise an operational setup that enables us to verify the (binary) session types of § 2 at runtime. Our runtime analysis is conducted by *uni-verdict* rejection monitors, whose purpose is to flag any session type violations detected (*i.e., violation monitors* [30, 3]).

3.1 Monitor and Instrumentation Design

We now illustrate the design decisions behind our formal monitoring framework. To this end, we use as a reference the client-server system outlined in Example 1. Consider, in particular, the scenario depicted in Fig. 5a, where a *client* is expected to interact with a *server* following the prescribed protocol S_{auth}; the *server* is *trusted* and guaranteed to adhere to the dual type $\overline{S_{auth}}$ (*e.g.,* because it has been statically typechecked against $\overline{S_{auth}}$ using the type system in § 2.3) – but we have limited control over the *client*, which might be untyped, hence its interactions are potentially unsafe.

Our setup should place no assumptions on the *client*, largely treating it as a black box. In fact, we target scenarios where the *client* source is inaccessible, possibly remote, interacting with the server via a generic channel of communication (*e.g.,* TCP sockets or HTTP addresses). This precludes the possibility of weaving the monitor within the *client* component. To achieve a model that can handle these requirements, we restrict ourselves to *outline monitors* [13, 7], which are decoupled from the process-under-scrutiny as concurrent units of code that can be more readily deployed over a black-box component; outline monitors are also easier to verify for correctness via compositional techniques [23, 26, 27, 14, 31, 28].



(a) No monitors.

(b) Server side instrumentation.

(c) Channel instrumentation.

Figure 5 Design choices for instrumentation.

Our model focusses on the communication occurring on the channel between the *client* and the *server* – and we assume such communication to be *synchronous* and *reliable*. Outline monitors can typically only analyse the externally *observable* actions of a monitored component. In our case, monitored processes follow the semantics of Fig. 2, hence the only observable actions are send ($\triangleleft \texttt{l}(v)$) and receive ($\triangleright \texttt{l}(v)$); τ-moves are unobservable.

We consider two potential instrumentation setup designs for an outline approach. In the setup in Fig. 5b, the *server* is instrumented with a *sequence-recogniser monitor* [57, 43] ($monitor_a$). The *server* is required to notify $monitor_a$ about every send and receive action it performs – this can be achieved via listeners added through mechanisms such as class-loaders, agents and VM-level tracers. For $monitor_a$, every receive action the *server* performs indicates a send action by the *client* and vice-versa (*i.e.,* every send indicates a receive). In Fig. 5b, The *client* sends the message `Auth("Bob","pwd")` to the *server* ①. Once received, ② the *server* notifies $monitor_a$ with the message contents and the direction of the message (▷). For $monitor_a$ this indicates that the *client* sent the particular message. After the *server* replies with the message `Fail(1)` ③, it notifies $monitor_a$ with the message contents and the direction (◁) ④, indicating that the *client* received the message.

In the alternative setup depicted in Fig. 5c, the monitor ($monitor_b$) is instrumented on the communication channel and acts as a *proxy* (or a *partial-identity monitor* [34]) between the two components. Any communicated messages must pass through $monitor_b$ in order for it to analyse them. In the execution of Fig. 5c the *client* sends the message `Auth("Bob","pwd")` to $monitor_b$ ①. The monitor checks that its contents conform with the protocol before proceeding to forward the message to the *server* ②. The *server* replies by sending the message `Fail(1)` to $monitor_b$ ③, which forwards it straight to the *client* ④.

On the one hand, the monitor in Fig. 5b is completely passive: it performs analysis in response to the events received. On the other hand, the monitor in Fig. 5c is also responsible for *forwarding* messages between the *client* and the *server*. Thus, the communication between the two components in Fig. 5c *relies* on $monitor_b$: should the monitor crash or terminate abruptly, the *client* and the *server* will stop interacting. Moreover, the setup in Fig. 5c introduces additional delays when every communicated message passes through $monitor_b$; these are avoided in Fig. 5b. The main drawback of the setup in Fig. 5b is that the *server* is directly exposed to an untrusted client, with additional responsibility of reporting events. In contrast, the instrumentation in Fig. 5c provides a layer of protection to the *server* from potentially malicious interactions: if the *client* sends a message that violates the protocol, $monitor_b$ is able to flag the message without forwarding it to the *server*. Moreover, the setup in Fig. 5c provides more flexibility for reasoning on the run-time monitoring of systems where *both* the *client* and the *server* are black boxes. This work opts for the setup in Fig. 5c.

3.2 A Monitor Calculus

Fig. 6 describes the structure and behaviour of a partial-identity monitor operating as in Fig. 5c. Monitors are similar to the processes defined in Fig. 2, with a few key additions. Since monitors need to interact with the environment, they also include the constructs $\blacktriangle\mathtt{l}(a).M$ and $\blacktriangledown\{\mathtt{l}_i(x_i : B_i).M_i\}_{i\in I}$, and rules [mOut] and [mIn]: they are analogous to the process output and input constructs, where interaction takes place between the environment and the monitor instead. We use the terms **internal** and **external** to differentiate between actions involving the monitored process and the environment, respectively.

As shown in Figure 7, monitors can reach two kinds of **rejection verdicts**, namely $\mathtt{no}_P$ and $\mathtt{no}_E$; the P and E tags distinguish between violations committed by the *monitored process* (P) and the *environment* (E). The rules [mIV] and [mEV] specify how the monitor reaches a verdict. Rule [mIV] represents the case when the monitor receives a violating message $\mathtt{l}(v)$ and consequently reaches the verdict $\mathtt{no}_P$; the message is deemed violating since its label is not among those that the monitor expects to receive. Symmetrically, in rule [mEV] the monitor reaches $\mathtt{no}_E$ when it receives a violating message from the external environment. The following example outlines the scenarios in which monitors reach a verdict.

Syntax

$$\text{Monitor}\quad M,N \coloneqq \triangleleft\mathtt{l}(a).M \mid \triangleright\{\mathtt{l}_i(x_i).M_i\}_{i\in I} \mid \blacktriangle\mathtt{l}(a).M \mid \blacktriangledown\{\mathtt{l}_i(x_i).M_i\}_{i\in I} \mid \mu_X.M \mid X \mid \mathsf{if}\ A\ \mathsf{then}\ M\ \mathsf{else}\ N \mid \mathbf{0} \mid \mathsf{no}_P \mid \mathsf{no}_E$$

Semantics

$$[\textsc{mSnd}]\ \frac{}{\triangleleft\mathtt{l}(v).M \xrightarrow{\triangleleft\mathtt{l}(v)} M} \qquad [\textsc{mOut}]\ \frac{}{\blacktriangle\mathtt{l}(v).M \xrightarrow{\blacktriangle\mathtt{l}(v)} M} \qquad [\textsc{mRec}]\ \frac{}{\mu_X.M \xrightarrow{\tau} M[\mu_X.M/X]}$$

$$[\textsc{mRcv}]\ \frac{j\in I}{\triangleright\{\mathtt{l}_i(x_i).M_i\}_{i\in I} \xrightarrow{\triangleright\mathtt{l}_j(v)} M_j[v/x_j]} \qquad [\textsc{mIn}]\ \frac{j\in I}{\blacktriangledown\{\mathtt{l}_i(x_i).M_i\}_{i\in I} \xrightarrow{\blacktriangledown\mathtt{l}_j(v_j)} M_j[v_j/x_j]}$$

$$[\textsc{mTru}]\ \frac{A \Downarrow \mathsf{tt}}{\mathsf{if}\ A\ \mathsf{then}\ M\ \mathsf{else}\ N \xrightarrow{\tau} M} \qquad [\textsc{mFls}]\ \frac{A \Downarrow \mathsf{ff}}{\mathsf{if}\ A\ \mathsf{then}\ M\ \mathsf{else}\ N \xrightarrow{\tau} N}$$

Violation Semantics

$$[\textsc{mIV}]\ \frac{\forall i\in I : \mathtt{l}\neq\mathtt{l}_i}{\triangleright\{\mathtt{l}_i(x_i).M_i\}_{i\in I} \xrightarrow{\triangleright\mathtt{l}(v)} \mathsf{no}_P} \qquad [\textsc{mEV}]\ \frac{\forall i\in I : \mathtt{l}\neq\mathtt{l}_i}{\blacktriangledown\{\mathtt{l}_i(x_i).M_i\}_{i\in I} \xrightarrow{\blacktriangledown\mathtt{l}(v)} \mathsf{no}_E}$$

Figure 6 Monitor Syntax and Semantics.

(a) Internal violation. **(b)** External violation.

Figure 7 Monitor violations.

▶ **Example 6.** Fig. 7 depicts a monitor verifying the conformity of a *client* with the session type S_{auth} (from Example 3). In Fig. 7a, the *client* sends the message `Login("Bob")`. Since the type S_{auth} states that the *client* should send a message with label `Auth`, the monitor reaches the verdict no_P by rule [mIV]. In Fig. 7b, the monitor receives `Res`(227) from the environment (which represents a buggy server). In this case the monitor reaches the verdict no_E (by rule [mEV]) since the message does not conform with S_{auth} which states that the *client* should receive either `Succ` or `Fail`. ⌟

▶ Remark 7. According to Fig. 6, our monitors can reach a verdict explicitly in their syntax (by having $\mathsf{no}_P/\mathsf{no}_E$ in their body), or by just transitioning to a verdict via rules [mIV] or [mEV]. We will make use both methods for our synthesised monitors (see § 3.4). ⌟

3.3 Composite Monitored System

The rules in Fig. 8 formalise the behaviour of the monitor when composed with the process to monitor, while also interacting with an environment (*i.e.,* another process). This setup is depicted in Fig. 9. We refer to a process P instrumented with a monitor M as a **composite (monitored) system**, denoted as $\langle P; M\rangle$. The rules [iRcv] and [iSnd] model the interaction *within* the composite system, (*i.e.,* between the monitored process P and the monitor M in Fig. 9). Note that the interaction between the two is *synchronous*: the monitor (resp. process) can only send a message when the process (resp. monitor) can receive the same message. If P sends a message (by [iSnd]) that violates the monitor's inputs, M is able to flag the violation by rule [mIV]. The rules [iOut] and [iIn] model the interaction between the composite system and the environment. As shown in Fig. 9, the monitor is the entity that interacts with the environment (represented as a process Q). Accordingly, the monitor can flag a message sent by the environment if the message violates the monitor's expected

$$[\textsc{iSnd}]\ \frac{P \xrightarrow{\triangleleft \mathtt{l}(v)} P' \quad M \xrightarrow{\triangleright \mathtt{l}(v)} M'}{\langle P; M\rangle \xrightarrow{\tau} \langle P'; M'\rangle} \qquad [\textsc{iRcv}]\ \frac{P \xrightarrow{\triangleright \mathtt{l}(v)} P' \quad M \xrightarrow{\triangleleft \mathtt{l}(v)} M'}{\langle P; M\rangle \xrightarrow{\tau} \langle P'; M'\rangle}$$

$$[\textsc{iOut}]\ \frac{M \xrightarrow{\blacktriangle \mathtt{l}(v)} M'}{\langle P; M\rangle \xrightarrow{\blacktriangle \mathtt{l}(v)} \langle P; M'\rangle} \qquad [\textsc{iIn}]\ \frac{M \xrightarrow{\blacktriangledown \mathtt{l}(v)} M'}{\langle P; M\rangle \xrightarrow{\blacktriangledown \mathtt{l}(v)} \langle P; M'\rangle}$$

$$[\textsc{iProc}]\ \frac{P \xrightarrow{\tau} P'}{\langle P; M\rangle \xrightarrow{\tau} \langle P'; M\rangle} \qquad [\textsc{iMon}]\ \frac{M \xrightarrow{\tau} M'}{\langle P; M\rangle \xrightarrow{\tau} \langle P; M'\rangle}$$

Figure 8 Composite monitored system semantics.

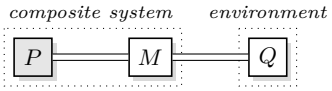


Figure 9 The composite system interacting with the environment.

inputs, by rule [mEV]. The rules [iProc] and [iMon] allow the monitored process and the monitor respectively to perform actions independent of each other (*e.g.,* to recurse or branch internally).

Our partial identity monitors halt upon reaching a verdict, in contrast to instrumented sequence recognisers that operationally continue to process events without changing their (irrevocable) verdict [26, 28]. As a result, our monitors also halt any interactions between the composite system and the environment. Because of this, monitor correctness is of paramount importance. The following example outlines the impact of a poorly constructed monitor.

▶ **Example 8.** Recall process $P_{auth} = \mu X. \triangleleft \mathtt{Auth}(\mathtt{"Bob"}, \mathtt{"pwd"}).P_{res}$ (Example 2), which adheres to the session type $S_{auth} = \mathsf{rec}\ Y.!\mathtt{Auth}(uname : \mathsf{Str}, pwd : \mathsf{Str}).S_{res}$ (Example 5). A monitor corresponding to S_{auth} should *receive* from P_{auth}, analyse the message, and forward it to the environment. The following (erroneous) monitor might seem to monitor S_{auth}:

$$M_{bad} = \triangleright\mathtt{Login}(uname).\blacktriangle\mathtt{Login}(uname).N_{bad}$$

If process P_{auth} is instrumented with monitor M_{bad}, we observe the following behaviour:

$$\langle P_{auth}; M_{bad}\rangle \xrightarrow{\tau} \langle \triangleleft\mathtt{Auth}(\mathtt{"Bob"}, \mathtt{"pwd"}).P_{res}[^{P_{auth}}/_X]; M_{bad}\rangle \xrightarrow{\tau} \langle P_{res}[^{P_{auth}}/_X]; \mathtt{no}_P\rangle$$

After P_{auth} unfolds, it sends the message $\mathtt{Auth}(\mathtt{"Bob"}, \mathtt{"pwd"})$ to the monitor as per S_{auth}. However, M_{bad} can only receive messages with label `Login`, hence it transitions to $\mathtt{no}_P$. ⌟

3.4 Monitor Synthesis

Def. 9 presents a synthesis procedure from session types (Fig. 3) to monitors (Fig. 6). The monitors generated are meant to act as a proxy between the monitored process and the environment process, as outlined in Fig. 5c. There are various practical advantages in having an automated synthesis function: it is less error prone, expedites development and improves the maintainability of the verification framework.

▶ **Definition 9.** *The monitor synthesis function* $[\![-]\!] : S \mapsto M$ *takes as input a session type* S *and returns a monitor* M*. It is defined inductively, on the structure of the session type* S*:*

$$[\![\oplus\{!\,\mathtt{l}_i(B_i).S_i\}_{i\in I}]\!] \triangleq \triangleright\{\mathtt{l}_i(x_i).\mathit{if}\ \mathit{isB}_i(x_i)\ \mathit{then}\ \blacktriangle\mathtt{l}_i(x_i).[\![S_i]\!]\ \mathit{else}\ \mathit{no}_P\}_{i\in I}$$

$$[\![\&\{?\,\mathtt{l}_i(B_i).S_i\}_{i\in I}]\!] \triangleq \blacktriangledown\{\mathtt{l}_i(x_i).\mathit{if}\ \mathit{isB}_i(x_i)\ \mathit{then}\ \triangleleft\,\mathtt{l}_i(x_i).[\![S_i]\!]\ \mathit{else}\ \mathit{no}_E\}_{i\in I}$$

$$[\![\mathit{rec}\ X.S]\!] \triangleq \mu_X.[\![S]\!] \qquad [\![X]\!] \triangleq X \qquad [\![\mathit{end}]\!] \triangleq \mathbf{0}$$ ⌟

The main cases of Def. 9 are those for the selection and branching types. In the case of $S = \oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}$, the synthesised monitor first waits to receive a message from the monitored process, with one of the labels specified in the type. Once the message is received, the monitor checks whether its payload is of the correct base type B_i, *i.e.*, $\mathsf{isB}_i(x_i)$ (see Remark 4), raising no_P if it is not. If $\mathsf{isB}_i(x_i)$ is true, the monitor forwards the message towards the environment, and proceeds according to $[\![S_i]\!]$. The synthesis for $S = \&\{?\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}$ is analogous, but the generated monitor receives a message from the environment, analyses it, and forwards it to the monitored process; any violations are attributed to the environment.

▶ **Example 10** (Session Monitor Synthesis)**.** Recall the session type S_{auth} in Example 3:

$$S_{auth} = \mathsf{rec}\ Y.!\mathtt{Auth}(\mathsf{Str},\mathsf{Str}).S_{res} \quad \text{where} \quad S_{res} = \&\{?\mathtt{Succ}(\mathsf{Str}).S_{succ}, ?\mathtt{Fail}(\mathsf{Int}).Y\}$$

The synthesis for this type first generates the recursion construct μ_Y followed by the synthesis for the selection type:

$$M_{auth} = [\![S_{auth}]\!] = \begin{cases} \mu_Y.\triangleright\,\mathtt{Auth}(uname,pwd).\ \mathsf{if}\ \big(\mathsf{isB}_{\mathsf{Str}}(uname)\wedge\mathsf{isB}_{\mathsf{Str}}(pwd)\big) \\ \qquad\qquad \mathsf{then}\ \blacktriangle\mathtt{Auth}(uname,pwd).[\![S_{res}]\!]\ \mathsf{else}\ \mathsf{no}_P \end{cases}$$

Monitor M_{auth} first waits to receive a message with label `Auth` from the monitored process (via $\triangleright$), checks the types of the payload $\big(\mathsf{isB}_{\mathsf{Str}}(uname)\wedge\mathsf{isB}_{\mathsf{Str}}(pwd)\big)$, and proceeds to forward the message to the environment (via $\blacktriangle$), continuing as the monitor of S_{res}:

$$[\![S_{res}]\!] = \blacktriangledown\left\{\begin{array}{l}\mathtt{Succ}(tok).\mathsf{if}\ \mathsf{isB}_{\mathsf{Str}}(tok)\ \mathsf{then}\ \triangleleft\,\mathtt{Succ}(tok).[\![S_{succ}]\!]\ \mathsf{else}\ \mathsf{no}_E,\\ \mathtt{Fail}(code).\mathsf{if}\ \mathsf{isB}_{\mathsf{Int}}(code)\ \mathsf{then}\ \triangleleft\,\mathtt{Fail}(code).Y\ \mathsf{else}\ \mathsf{no}_E\end{array}\right\}$$

Observe that $[\![S_{res}]\!]$ inputs from the environment and outputs to the monitored process. ⌟

If process P_{auth} is instrumented with monitor M_{auth} as the composite system $\langle P_{auth}; M_{auth}\rangle$, we observe the behaviour outlined in Fig. 5c, as we show in the following example.

▶ **Example 11.** Recall P_{auth} defined in Example 2:

$$P_{auth} = \mu_X.(\triangleleft\mathtt{Auth}(\texttt{"Bob"},\texttt{"pwd"})).P_{res} \quad \text{where} \quad P_{res} = \triangleright\{\mathtt{Succ}(tok).P_{succ}, \mathtt{Fail}(code).P_{fail}\}$$

When P_{auth} is instrumented with the monitor $M_{auth} = [\![S_{auth}]\!]$ we observe the behaviour:

$$\langle P_{auth}; M_{auth}\rangle \xrightarrow{\tau} \langle P'_{auth}; M_{auth}\rangle \qquad \text{where } P'_{auth} = \triangleleft\mathtt{Auth}(\texttt{"Bob"},\texttt{"pwd"}).P_{res}[^{P_{auth}}/_X]$$

$$\langle P'_{auth}; M_{auth}\rangle \xrightarrow{\tau} \langle P'_{auth}; M'_{auth}\rangle \qquad \text{using [IMON]}$$
$$\text{where } M'_{auth} = \big(\triangleright\,\mathtt{Auth}(uname,pwd).\mathsf{if}\ \big(\mathsf{isB}_{\mathsf{Str}}(uname)\wedge\mathsf{isB}_{\mathsf{Str}}(pwd)\big) \ \mathsf{then}\ \blacktriangle\mathtt{Auth}(uname,pwd).[\![S_{res}]\!]\ \mathsf{else}\ \mathsf{no}_P\big)[^{M_{auth}}/_Y]$$

After unfolding, using the rules [IPROC] and [IMON] respectively, the monitor can receive and the process can send, and they can transition together to communicate: (see ① in Fig. 5c)

$$P'_{auth} \xrightarrow{\triangleleft\mathtt{Auth}(\texttt{"Bob"},\texttt{"pwd"})} P''_{auth} \text{ where } P''_{auth} = \triangleright\{\mathtt{Succ}(tok).P_{succ}, \mathtt{Fail}(code).P_{fail}\}[^{P_{auth}}/_X]$$

$$M'_{auth} \xrightarrow{\triangleright\mathtt{Auth}(\texttt{"Bob"},\texttt{"pwd"})} M''_{auth} \text{ where}$$

$$M''_{auth} = \mathsf{if}\ \big(\mathsf{isB}_{\mathsf{Str}}(\texttt{"Bob"})\wedge\mathsf{isB}_{\mathsf{Str}}(\texttt{"pwd"})\big)\ \mathsf{then}\ \blacktriangle\mathtt{Auth}(\texttt{"Bob"},\texttt{"pwd"}).[\![S_{res}]\!][^{M_{auth}}/_Y]\ \mathsf{else}\ \mathsf{no}_P$$

$$\langle P'_{auth}; M'_{auth}\rangle \xrightarrow{\tau} \langle P''_{auth}; M''_{auth}\rangle$$

The monitor proceeds by checking the values of the payload values using the rule [IMON].

$$M''_{auth} \xrightarrow{\tau} M'''_{auth} \quad \text{where } M'''_{auth} = \blacktriangle\mathtt{Auth}(\texttt{"Bob"},\texttt{"pwd"}).[\![S_{res}]\!][^{M_{auth}}/_Y]$$

$$\langle P''_{auth}; M''_{auth}\rangle \xrightarrow{\tau} \langle P''_{auth}; M'''_{auth}\rangle$$

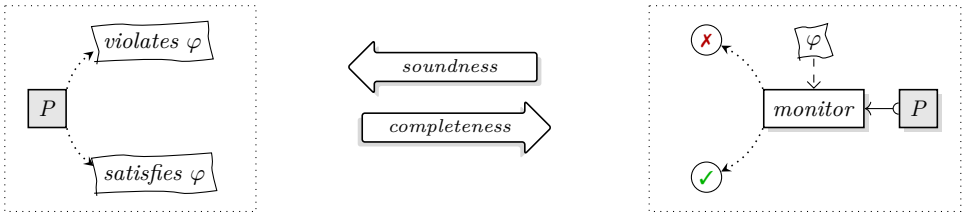


Figure 10 Monitoring soundness and completeness, from a logic-based viewpoint [30, 3, 4].

M'''_{auth} now forwards the message to the environment by rule [IOUT]: (see ② in Fig. 5c)

$$\langle P''_{auth}; M'''_{auth}\rangle \xrightarrow{\blacktriangle\texttt{Auth("Bob","pwd")}} \langle P''_{auth}; [\![S_{res}]\!][M_{auth}/Y]\rangle$$

The monitor is currently waiting to receive from the environment, since:

$$[\![S_{res}]\!][M_{auth}/Y] = \blacktriangledown \left\{ \begin{array}{l} \texttt{Succ}(tok).\mathsf{if}\ \mathsf{isB}_{\mathsf{Str}}(tok)\ \mathsf{then} \triangleleft \texttt{Succ}(tok).[\![S_{succ}]\!][M_{auth}/Y]\ \mathsf{else}\ \mathsf{no}_E \\ \texttt{Fail}(code).\mathsf{if}\ \mathsf{isB}_{\mathsf{Int}}(code)\ \mathsf{then} \triangleleft \texttt{Fail}(code).M_{auth}\ \mathsf{else}\ \mathsf{no}_E \end{array} \right\}$$

If the monitor receives the message `Succ(321)`, it forwards the message to the monitored process and proceeds according to $[\![S_{succ}]\!][M_{auth}/Y]$. If the monitor receives the message `Fail(1)` (see ③ in Fig. 5c) it forwards the message to the process P''_{auth} (see ④ in Fig. 5c):

$$\triangleleft\texttt{Fail(1)}.M_{auth} \xrightarrow{\triangleleft\texttt{Fail(1)}} M_{auth} \qquad P''_{auth} \xrightarrow{\triangleright\texttt{Fail(1)}} P_{fail}[P_{auth}/X][1/code]$$
$$\langle P''_{auth}; \triangleleft\texttt{Fail(1)}.M_{auth}\rangle \xrightarrow{\tau} \langle P_{fail}[P_{auth}/X][1/code]; M_{auth}\rangle$$

The composite system can now proceed with the monitor restarting as M_{auth}. ⌟

Should the monitored process send a message that violates the session type, the monitor can flag the violation upon receiving a message, as the following example shows.

▶ **Example 12.** Consider the scenario in Fig. 7a, where the *client* is the process P_{bad}:

$$P_{bad} = \triangleleft\texttt{Login("Bob")}.\triangleright\texttt{Res}(tok : \mathsf{Str}).P_{res}$$

and recall the monitor M_{auth} (from Examples 10 and 11) obtained from the session type S_{auth}. When P_{bad} is instrumented with M_{auth}, we observe the following behaviour:

$$\langle P_{bad}; M_{auth}\rangle \xrightarrow{\tau} \langle P_{bad}; M'_{auth}\rangle \xrightarrow{\tau} \langle \triangleright\texttt{Res}(tok : \mathsf{Str}).P_{res}; \mathsf{no}_P\rangle \qquad \text{using [IMON],[ISND]}$$

$$\text{where } M'_{auth} = \left\{ \begin{array}{l} \big(\triangleright\,\texttt{Auth}(uname, pwd).\mathsf{if}\ \big(\mathsf{isB}_{\mathsf{Str}}(uname) \wedge \mathsf{isB}_{\mathsf{Str}}(pwd)\big) \\ \qquad\qquad \mathsf{then}\ \blacktriangle\texttt{Auth}(uname, pwd).[\![S_{res}]\!]\mathsf{else}\ \mathsf{no}_P\big)[M_{auth}/Y] \end{array} \right.$$

i.e., M_{auth} unfolds, receives `Login("Bob")` from P_{bad}, and flag the rejection verdict no_P. ⌟

4 Formal Analysis and Results

In § 3 we argued for the importance of monitor correctness. This has also been recognised by other works that study monitoring techniques for session types [17, 42, 34]. However, these attempts all propose their own bespoke notion of monitor correctness that is often hard to relate to the others. Instead, we strive towards a more systematic approach for monitor correctness and study monitor correctness in relation to an independent characterisation of process correctness. More concretely, we assess the correctness of *session monitors* in relation to *session typing*. We draw inspirations from a recent body of work that captures this

relationship in terms of *soundness* and *completeness* [30, 3], as depicted in Fig. 10. In such body of work, *monitor soundness* states that if a monitor M is monitoring a process P for a property φ, and M reaches a rejection (resp. acceptance) verdict, then such a verdict must correspond to P's violations (resp. satisfactions) of property φ. *Monitor completeness* is the dual property: if a process P violates (resp. satisfies) a property φ, then the monitor that runtime-checks P for φ must reach a rejection (resp. acceptance) verdict. This formulation is appealing to our study for a number of reasons:

- The touchstone logic used to specify process correctness is the Hennessy-Milner Logic with minimal and maximal fixpoints (recHML) [45]; like session types, it has a tight relation to (ω-)regular properties, and a long tradition of automata-based interpetations.
- Recent work [4, 6] has extended this framework to a spectrum of correctness criteria. This gives us the flexibility of identifying the criteria that best fit our concerns.

To study session types monitorability, we adapt this theoretical framework to our setting:

M1 instead of logic formulas as specifications, we adopt session types as specifications; and

M2 to characterise processes satisfying a specification, we use the session typing system.

This leads to important differences between our approach and [30, 3]:

D1 by item M2, our processes characterisation is syntactic (rather than semantic), which is further removed from the runtime behaviour observed by the monitor;

D2 session types describe interactions between two parties, and our monitors can attribute a violation to a party. By contrast, monitors for recHML formulas flag generic rejections;

D3 we here limit our analysis to rejection monitors and do not consider acceptance verdicts.

Consequently, we formalise our notions of monitoring soundness and completeness as follows. Here, t represents a *trace*, *i.e.,* finite a sequence of environment send/receive actions $\blacktriangle\mathtt{l}(v)$ and $\blacktriangledown\mathtt{l}(v)$ (from Fig. 8); moreover, $\stackrel{t}{\Rightarrow}$ is a sequence of transitions where the actions in t are interleaved with finite sequences of τ-transitions.

▶ **Definition 13** (*Session Monitor Soundness*). *A monitor M* soundly monitors *for a session type S iff, for all P, if there is a trace t such that $\langle P; M\rangle \stackrel{t}{\Rightarrow} \langle P'; \mathsf{no}_P\rangle$, then $\emptyset\cdot\emptyset \vdash P : S$ does* not *hold.* ⌟

▶ **Definition 14** (*Session Monitor Completeness*). *A monitor M monitors for a session type S in a* complete *manner, iff for all processes P, whenever $\emptyset\cdot\emptyset \vdash P : S$ does* not *hold, then there exists a trace t such that $\langle P; M\rangle \stackrel{t}{\Rightarrow} \langle P'; \mathsf{no}_P\rangle$.* ⌟

4.1 Soundness of Session Type Monitoring

A tenet of [30, 3, 4] is that, in order to have monitor correctness, soundness (Def. 13) is not negotiable. We here show that our monitor synthesis procedure is sound, *i.e.,* we show that for any session type S, monitor $[\![S]\!]$ observes Def. 13 w.r.t. specification S.

▶ **Theorem 15** (*Synthesis Soundness*). *For all session types S and processes P, if there exists a trace t such that $\langle P; [\![S]\!]\rangle \stackrel{t}{\Rightarrow} \langle P'; \mathsf{no}_P\rangle$, then $\emptyset\cdot\emptyset \vdash P : S$ does* not *hold.*

Proof. Instead of proving the statement directly, we prove its contrapositive:

> For all session types S and processes P such that $\emptyset\cdot\emptyset \vdash P : S$, if $\langle P; [\![S]\!]\rangle \stackrel{t}{\Rightarrow} \langle P'; M'\rangle$ then $M' \neq \mathsf{no}_P$.

To this end, we first establish a *subject reduction* result, relying on standard properties of our type system: this determines how process P evolves w.r.t. its session type S. Then, we prove the contrapositive statement above by *lexicographical induction* on the derivation of

$\emptyset \cdot \emptyset \vdash P : S$ and the number of transitions in the trace $\langle P; [\![S]\!] \rangle \stackrel{t}{\Rightarrow} \langle P'; M' \rangle$. This requires some sophistication, because as the instrumented system $\langle P; [\![S]\!] \rangle$ evolves, for each step of P the monitor $[\![S]\!]$ (as generated by Def. 9) may take multiple steps to evaluate synthesised conditions before it can forward messages. Hence, we prove additional results to handle such cases, and formulate a suitable induction hypothesis allowing us to complete the proof of the contrapositive statement. Theorem 15 follows as a corollary. ◀

As a by-product of Theorem 15 we also deduce that if a process P has type S, then the instrumented process $\langle P; [\![S]\!] \rangle$ can only get stuck due to an *external* violation, *i.e.,* no_E; this arises when the environment sends a message with a wrong label or payload type. This result is formalised in Corollary 16 below, and is reminiscent of the notion of *blaming* in gradual types (*i.e.,* untyped components can always be blamed in case of errors [10, 41]).

▶ **Corollary 16** (Monitor Blaming). *For any process P and session types S where $\emptyset \cdot \emptyset \vdash P : S$, for any trace t such that $\langle P; [\![S]\!] \rangle \stackrel{t}{\Rightarrow} \langle P'; M' \rangle \not\rightarrow$ where $P \neq \mathbf{0}$, we have $M' = \mathsf{no}_E$.* ⌟

4.2 On the Completeness of Session Type Monitoring

Monitor soundness, by itself, is a weak result. For instance, the monitor that merely acts as a forwarder between the monitored process and the environment, *never raising any detections*, is trivially sound but, arguably, not very useful. One way to force the monitor to produce useful detections is via *completeness*, as per Def. 14 above. We investigate completeness for our synthesised monitors by first establishing a "weak" completeness result (§ 4.2.1) showing how ill-typed processes can misbehave when instrumented. Then, we prove that a "full" completeness result is impossible in our black-box monitoring model (§ 4.2.2).

4.2.1 Weak Monitor Synthesis Completeness

To achieve our completeness result, in this section we need a *precise typing* assumption on predicates A: ill-typed predicates do not evaluate to a boolean – *i.e.,* if $\Gamma \vdash A : \mathsf{Bool}$ does *not* hold, then $A \not\Downarrow \mathsf{tt}$ and $A \not\Downarrow \mathsf{ff}$. Furthermore, we need to limit our analysis to processes without *dead code* (Def. 18 below). For the process language of Fig. 2, this means: for every "if" statement occurring in a process P, there are executions of P where the left branch is taken, and executions where the right branch is taken. These executions depend on P's inputs, which may cause different instantiations to P's variables. Example 17 illustrates why we need this assumption; note that these assumptions are *not* needed for monitor soundness.

▶ **Example 17.** The process $P = \mathsf{if}\ \mathsf{tt}\ \mathsf{then}\ \triangleleft \mathtt{l}_1(v_1).\mathbf{0}\ \mathsf{else}\ \triangleleft \mathtt{l}_2(v_2).\mathbf{0}$ is *not* typable with $S = \oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i \in \{1\}}$ (for any S_i): it is only typable with internal choices of the form $\oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i \in 1..n}$, with $n \geq 2$. Yet, P would operate correctly if instrumented with monitor $[\![S]\!]$, because its "else" branch is dead code. If we remove the dead code from P, the remaining process $\triangleleft \mathtt{l}_1(v_1).\mathbf{0}$ is typable with S, and behaves like P. ⌟

▶ **Definition 18.** A process P has no dead code *iff for all its subterms of the form $P' = \mathsf{if}\ A\ \mathsf{then}\ Q\ \mathsf{else}\ Q'$, there exist traces t and t' and substitutions σ and σ' such that $P \stackrel{t}{\Rightarrow} P'\sigma \stackrel{\tau}{\rightarrow} Q\sigma$ (hence, $A\sigma \Downarrow \mathsf{tt}$) and $P \stackrel{t'}{\Rightarrow} P'\sigma' \stackrel{\tau}{\rightarrow} Q'\sigma'$ (hence, $A\sigma \Downarrow \mathsf{ff}$).* ⌟

With the "no dead code" assumption, we can formulate our weak completeness result. It states that when a process P is ill-typed for a session type S, then the monitored system $\langle P; [\![S]\!] \rangle$ exhibits at least one execution that gets stuck due to P's behaviour, without any violation by the environment.

▶ **Theorem 19** (Weak Monitor Synthesis Completeness). *Take any closed process P without dead code such that $\emptyset \cdot \emptyset \vdash P : S$ does* not *hold. Then, there exists a trace t such that $\langle P; [\![S]\!] \rangle \stackrel{t}{\Rightarrow} \langle P'; M' \rangle \not\rightarrow$, with $P' \neq \mathbf{0}$ or $M' \neq \mathbf{0}$; moreover, $M' \neq \mathsf{no}_E$.*

Proof. The proof is based on *failing derivations*, inspired by [16, 44]. It consists of 6 steps.

1. We define the *rule function* Φ that, following the typing rules in Fig. 4, maps a judgement of the form $J = \Theta \cdot \Gamma \vdash P : S$ to either the set of all judgements in J's premises (for inductive rules), or $\{\mathsf{tt}\}$ (for axioms), or $\emptyset$ (if J does not match any rule);
2. we formalise a *failing derivation* of a session typing judgement $\Theta \cdot \Gamma \vdash P : S$ as a finite sequence of judgements $\mathcal{D} = (J_0, J_1, \ldots, J_n)$ such that:
   - **(i)** for all $i \in 0..n$, J_i is a judgement of the form $\Theta_i \cdot \Gamma_i \vdash P_i : S_i$;
   - **(ii)** $J_0 = \Theta \cdot \Gamma \vdash P : S$ (*i.e.,* the failing derivation $\mathcal{D}$ begins with the judgement of interest);
   - **(iii)** $\forall i \in 1..n$, $J_i \in \Phi(J_{i-1})$ (*i.e.,* each judgement in $\mathcal{D}$ is followed by one of its premises);
   - **(iv)** $\Phi(J_n) = \emptyset$ (*i.e.,* the last judgement in $\mathcal{D}$ does not match any rule in Fig. 4)
3. we prove there is a failing derivation of $J = \Theta \cdot \Gamma \vdash P : S$ if and only if J is *not* derivable;
4. we formalise a *negated* typing judgement $\Theta \cdot \Gamma \nvdash P : S$ and prove that it holds if and only if there is a corresponding failing derivation of $\Theta \cdot \Gamma \vdash P : S$;
5. thus, from items 3 and 4 above, we know that $\Theta \cdot \Gamma \vdash P : S$ is *not* derivable if and only if $\Theta \cdot \Gamma \nvdash P : S$ is derivable. Consequently, the judgement $\Theta \cdot \Gamma \nvdash P : S$ tells us exactly what are the possible shapes of P and S covered by the theorem's statement;
6. finally, we use all ingredients above to prove the thesis. From a failing derivation of $\Theta \cdot \Gamma \vdash P : S$ (item 3), we construct a trace t leading from $\langle P; M \rangle$ to some $\langle P'; M' \rangle$; further, using the corresponding derivation of $\Theta \cdot \Gamma \nvdash P : S$ (items 4, 5), we prove that t is a valid trace, and $\langle P'; M' \rangle \not\rightarrow$ with $P' \neq \mathbf{0}$ or $M' \neq \mathbf{0}$, and $M' \neq \mathsf{no}_E$. ◀

Although Theorem 19 is weaker than the ideal requirement set out in Def. 14, its proof sheds light on all the possible reasons why an ill-typed monitored process gets stuck:

1. the monitor reaches a process rejection verdict, $M' = \mathsf{no}_P$, because the process sends a message with a wrong label, or payload value of a wrong base type.
2. the monitor blocks waiting for the process to send a message, but:
   - **a.** P' is attempting to receive a message itself or
   - **b.** $P' = \mathbf{0}$ (i.e., P' has terminated its execution);
3. the monitor blocks waiting for the process to receive a message, but:
   - **a.** the process is also waiting to receive a message but does not support the required message label being sent or
   - **b.** P' is attempting to send a message itself or
   - **c.** $P' = \mathbf{0}$;
4. the monitor expects the process to end, but P' is trying to send/receive more messages;
5. P' is stuck on an ill-typed expression.

▶ Remark 20. Process violations are only flagged no_P (as required in Def. 14) is case 1. We now discuss how a practical monitor implementation could, in principle, detect violations in other cases, and highlight when this additional detection power would require additional assumptions that go beyond our black-box monitoring design.

- In cases 3a and 5, the trace t may lead to a run-time error; this could be flagged by assuming that the monitor can detect whether the monitored process has crashed;

- In case 4, the monitor expects the session to be ended. This could be handled by assuming and end-of-session signal: the monitor can wait for such a signal, and flag any other message sent by the process. However, if the process is attempting to receive (instead of ending the session), the detection is more subtle, as in case 2a below;
- Cases 2b and 3c could be similarly handled by assuming an end-of-session signal;
- Case 2a is more subtle: both the process and monitor are waiting for a message. Reception timeouts from the monitor side are inadequate because they lead to unsound detections. To accurately handle this case, we would need to instrument the process executable, which breaks our black-box assumptions from § 3.1. Similarly, flagging a violation in case 3b also requires access to the process code, again breaking our black-box design. ⌟

4.2.2 Impossibility of Sound and Complete Session Monitoring

The weakness of our completeness result in Theorem 19 is not specific to our monitor synthesis function. Rather, we show that this is an inherent limit of the operational model (Figures 6 and 8) that captures the black-box monitor design decisions of § 3.1. Similar impossibility results often arise for reasonably expressive specification languages (such as the logics in [30, 1, 3, 4]), where it is usually the case that only a subset of specifications can be monitored in a sound and complete way.

▶ **Theorem 21** (Impossibility of Sound and Complete Session Monitoring). *A (closed) session type $S \neq \mathsf{end}$ cannot have a sound and complete monitor under the semantics of Fig. 6.*

Proof. We proceed by case analysis on the structure of S:

Case $S = \&\{?\mathtt{l}_i(\mathbf{B}_i).S_i\}_{i\in I}$: We assume that a complete monitor M for S exists and proceed to show that such a monitor is necessarily unsound for S. Fix a complete monitor M for S. Consider the process $P_2 = \triangleright\{\mathtt{l}_i(x_i).Q_i\}_{i\in I}$ that is well-typed w.r.t. the session type S. Then, consider the process P_1 obtained by pruning some of the top-level external choices of P_2, *i.e.,* $P_1 = \triangleright\{\mathtt{l}_j(x_j).Q_j\}_{j\in J}$ where $J \subset I$ (a strict inclusion). Observe that P_1 is ill-typed for S, and thus, by completeness (Def. 14), M should reject P_1, (*i.e.,* there must exists a trace t such that $\langle P_1; M\rangle \stackrel{t}{\Rightarrow} \langle P_1'; \mathsf{no}_P\rangle$). There are two ways for M to reach such a verdict:

- $M \stackrel{t}{\Rightarrow} \mathsf{no}_P$ without interacting with P_1. In this case, the same rejection verdict is reached by the composite system $\langle P_2; M\rangle$. Since P_2 is well-typed for S, this means that M is unsound for S by Def. 13;
- M reaches the rejection verdict after interacting (at least once) with P_1. In this case, we have $P_1 \xrightarrow{\triangleright\mathtt{l}_j(v)} Q_j$ (for some $j \in J$), and there are t_1, t_2, P_1' such that $t = t_1.t_2$ and $M \xRightarrow{t_1.\triangleleft\mathtt{l}_j(v)} M'$ and $\langle Q_j; M'\rangle \stackrel{t_2}{\Longrightarrow} \langle P_1'; \mathsf{no}_P\rangle$. But then, since $j \in J \subseteq I$, we also have $\langle P_2; M\rangle \stackrel{t}{\Rightarrow} \langle P_1'; \mathsf{no}_P\rangle$. Since M rejects the well-typed process P_2, this again makes M unsound for S by Def. 13.

We have thus shown that a complete monitor M for S is necessarily unsound.

Case $S = \oplus\{!\mathtt{l}_i(\mathbf{B}_i).S_i\}_{i\in I}$: Assume that a complete monitor M for S exists. The process $P_1 = \mathbf{0}$ is ill-typed for S (since it does not produce any of the expected outputs). By Def. 14 (Completeness), there must exist a trace t such that $\langle P_1; M\rangle \stackrel{t}{\Rightarrow} \langle P_1'; \mathsf{no}_P\rangle$. From the structure of P_1 it is clear that M reaches its rejection verdict without interacting with P_1, *i.e.,* $M \stackrel{t}{\Rightarrow} \mathsf{no}_P$. This also means that M would also reach a rejection verdict when instrumented with $P_2 = \triangleleft\mathtt{l}_k(v_k).Q_2'$ with $k \in I$ and is well-typed w.r.t. S. This makes M unsound by Def. 13.

Recall that all session types are assumed to be guarded. Since the above two cases rule out all the guarding constructs, $\oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}$ and $\&\{?\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}$, we conclude that there is no closed (guarded) session type that can be monitored for soundly and completely, except for all the trivial session types that equate to end. ◀

5 Realisability and Implementation

Up to this point we have considered a level of abstraction that allows us to model session monitors and monitored components, reason about their behaviour, and prove their properties. We now illustrate how our theoretical developments can be translated into an actual implementation of session monitoring, targeting the Scala programming language. The key idea is to turn our monitor synthesis procedure (Def. 9) into a code generation tool that, given a protocol specification (as a session type), produces the Scala code of a corresponding executable monitor. The tool is called `STMonitor`, and is provided as companion artifact to this paper. It is also available at:

`https://github.com/chrisbartoloburlo/stmonitor` (release tag `v0.0.1`)

We describe `STMonitor` in § 5.2 – but first, we augment session types with *assertions* (§ 5.1).

5.1 Introducing Assertions in Session Types Specifications

Since we use session types as specifications for a tool that generates executable monitors, it is convenient to enrich them with *assertions* on the values being sent or received. We augment the session types syntax (Fig. 3) by extending selection and branching as follows:

$$S \;::=\; \oplus\{!\mathtt{l}_i(x_i:\mathsf{B}_i)[A_i].S_i\}_{i\in I} \;\mid\; \&\{?\mathtt{l}_i(x_i:\mathsf{B}_i)[A_i].S_i\}_{i\in I} \;\mid\; \dots$$

The assertions A_i are predicates of the process calculus (Fig. 2, Remark 4), and they can refer to the named payload variables x_i. Such assertions do not influence type-checking: they are copied in the synthesised monitors, where they are used to flag the new violations no^A_P (assertion violation by the process) and no^A_E (external assertion violation). To achieve this, we update our monitor synthesis function (Def. 9) as follows:

$$[\![\oplus\{!\mathtt{l}_i(x_i:\mathsf{B}_i)[A_i].S_i\}_{i\in I}]\!] \triangleq \triangleright\{\mathtt{l}_i(x_i).\mathsf{if}\ \mathsf{is}\mathsf{B}_i(x_i)\ \mathsf{then}\ (\mathsf{if}\ A_i\ \mathsf{then}\ \blacktriangle\mathtt{l}_i(x_i).[\![S_i]\!]\ \mathsf{else}\ \mathsf{no}^A_P)\ \mathsf{else}\ \mathsf{no}_P\}_{i\in I}$$

$$[\![\&\{?\mathtt{l}_i(x_i:\mathsf{B}_i)[A_i].S_i\}_{i\in I}]\!] \triangleq \blacktriangledown\{\mathtt{l}_i(x_i).\mathsf{if}\ \mathsf{is}\mathsf{B}_i(x_i)\ \mathsf{then}\ (\mathsf{if}\ A_i\ \mathsf{then}\ \triangleleft\mathtt{l}_i(x_i).[\![S_i]\!]\ \mathsf{else}\ \mathsf{no}^A_E)\ \mathsf{else}\ \mathsf{no}_E\}_{i\in I}$$

The only changes are highlighted: if the monitored process sends a message that violates the assertion, it is flagged with no^A_P; symmetrically, if a message that violates the assertion is received from the environment, then the message is flagged with no^A_E.

▶ **Example 22.** Recall S_{auth} from Example 3. We can refine it with assertions to check the validity of the data being transmitted and received:

$$\begin{aligned} S^A_{auth} &= \mathsf{rec}\ Y.!\mathtt{Auth}(\mathit{uname}:\mathsf{Str},\mathit{pwd}:\mathsf{Str})[\mathit{validUname}(\mathit{uname})].S^A_{res} \\ S^A_{res} &= \&\{?\mathtt{Succ}(\mathit{tok}:\mathsf{Str})[\mathit{validTok}(\mathit{tok},\mathit{uname})].S^A_{succ}, ?\mathtt{Fail}(\mathit{code}:\mathsf{Int})[\mathtt{tt}].Y\} \end{aligned}$$

In S^A_{auth}, when the *client* sends $\mathtt{Auth}(\mathit{uname},\mathit{pwd})$, the value of *uname* is passed to the predicate *validUname* which ensures that the supplied *uname* is given in the correct format. If the *server* replies with $\mathtt{Succ}(\mathit{tok})$, the token *tok* and username *uname* are validated by the cryptographic predicate *validTok*, which tests whether the token is correct for the given username. If so, the *client* continues along session type S^A_{succ}. Otherwise, if the *server* chooses to send `Fail` with the error *code*, the trivial assertion check `tt` is performed. ⌟

Notice that, when all assertions are trivially true, the augmented monitor synthesis is equivalent to the original Def. 9. Otherwise, the synthesised monitors with assertions are more restrictive: executions where no violations no_P nor no_E were detected might now violate an assertion and result in no_P^A or no_E^A. The introduction of such assertions in our theory changes our monitorability results as follows:

- soundness (Theorem 15) is preserved – which is crucial for practical usability;
- blaming (Corollary 16) is weakened: an instrumented well-typed process may violate an assertion, and be flagged with no_P^A;
- weak detection completeness (Theorem 19) is *not* preserved: assertions can in principle be unsatisfiable, hence some ill-typed processes may not be flagged because all their traces end with an environment assertion violation no_E^A.

5.2 Implementation

We now illustrate the implementation of our session monitor synthesis tool. It generates runnable Scala code from session types, possibly including the assertions discussed in § 5.1.

Implementation framework. Our synthesised monitors uses the session programming library `lchannels` [56]. It allows for implementing a session type S in Scala, by

1. defining a set of *Continuation-Passing-Style Protocol classes* (CPSPc) corresponding to S, and
2. using a communication API that, by leveraging such CPSPc, lets the Scala compiler spot protocol violations.

By using `lchannels`, we are more confident that if a syntesised monitor for session type S compiles, then it correctly sends/receives messages according to S. Moreover, `lchannels` abstracts communication from the underlying message transport, hence it allows our monitors to interact with clients or servers written in any programming language.

Implementation of the session monitor synthesis. Overall, our Scala monitor generation requires 3 user-supplied inputs:

i1 a session type S (with or without assertions) describing the desired protocol;

i2 for each assertion in S (if any), a corresponding Scala function returning true/false; and

i3 a *Connection Manager* class (discussed below) to interact with the monitored process.

Given a session type (input 1) our monitor synthesiser tool generates:

1. the protocol classes (CPSPc) for representing the session type in Scala + `lchannels`, and
2. the Scala source code of a runtime monitor (requiring inputs 2 and 3 to compile).

The generated monitor acts as a mediator between client and server: one is on the *internal* side of the monitor (*i.e.,* the instrumented process), while the other is on the external side. The internal side is untrusted: its messages are run-time checked, to ensure they follow the desired protocol (*e.g.,* session type S^A_{auth} in Example 22). Instead, the *external* side is trusted: it is (mostly) expected to follow the dual protocol (*e.g.,* the dual session type $\overline{S^A_{auth}}$). This design choice allows us to simplify the monitor implementation, as its communication with the external side are handled by `lchannels`. However, our design does not limit the flexibility of the approach, since an untrusted peer can be made trusted by instrumenting it with a monitor (see discussion below).

Monitor synthesis in practice. We now illustrate the scenario depicted in Fig. 11 where:

1. we want a client/server system to implement the session type (with assertions) S^A_{auth} (Example 22);
2. we trust the *server* (*e.g.,* because it is type-checked), and
3. we want to instrument a *client* whose source code is inaccessible or cannot be verified.

Other variations of this scenario are possible. For instance, we could similarly instrument an untrusted server, by running our monitor synthesiser on the dual session type $\overline{S_{auth}}$. The resulting combination of monitor-and-server is then trusted and can interact via `lchannels`. As a result, it could then be used as the trusted *server* in Fig. 11.

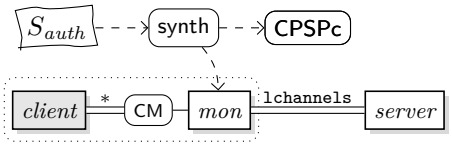


Figure 11 The composite system interacting with the environment.

The generated monitor (*mon*) intercepts all messages between *client* and *server*. The communication between *mon* and *server* occurs via `lchannels`; instead, the communication between the monitor and the client is handled by a *Connection Manager* (CM): a user-supplied Scala class, input 3, which acts as a *translator* and *gatekeeper*, by transforming each messages from the monitor-client transport protocol into a corresponding CPSP class, and *vice versa*. With this design, the code generated for the monitor is abstracted from the low-level details of the protocols used by both the *client* and *server*.

There is a tight correspondence between the monitors generated by our tool, and our formal monitor synthesis. This increases our confidence that the results in § 4 carry over to our implementation and that our tool is indeed correct. In the sequel, we illustrate the generated monitoring code for Example 22 above, showing the monitoring of a selection type (§ 5.2) and branching type (§ 5.2).

$$
\begin{aligned}
&[\![S^A_{auth}]\!] \;=\\
&\quad \mu_Y \,.\, \triangleright \,\{\mathtt{Auth}(\mathit{uname} : \mathsf{Str}, \mathit{pwd} : \mathsf{Str}).\\
&\quad \text{if } \mathsf{isB}_{\mathsf{Str}}(\mathit{uname}) \wedge \mathsf{isB}_{\mathsf{Str}}(\mathit{pwd})\\
&\quad \text{ then if } \mathit{validUname}(\mathit{uname})\\
&\quad\quad \text{then } \blacktriangle\mathtt{Auth}(\mathit{uname}, \mathit{pwd}).[\![S_{res}]\!]\\
&\quad\quad \text{else } \mathsf{no}^A_P\\
&\quad \text{else } \mathsf{no}_P \,\}
\end{aligned}
$$

```
def receiveAuth(srv:Out[Auth],client:CM): Unit ={
  client.receive() match {
    case msg @ Auth(_, _) =>
      if (validUname(msg.uname)) {
        val cont = srv !! Auth(msg.uname,
                               msg.pwd)_
        payloads.Auth.uname = msg.uname
        sendChoice1(msg.cont, client)
      } else {
        /* INTERNAL VIOLATION (assertion) */
      }
    case _ =>
      /* INTERNAL VIOLATION: invalid message */
} }
```

Figure 12 Comparison between the formal and implementation synthesis of the internal choice.

The internal receive operator of the monitor calculus ($\triangleright$) corresponds to line 2 in § 5.2, where the monitor invokes the `receive` method of the CM. Depending on the type of message received, the monitor performs a series of checks. By default, a catch-all case (line 12) handles

any messages violating the protocol: this is similar to rule [MIV] of the formal monitor (Fig. 6), which flags the violation $\mathtt{no}_P$. If `Auth` is received, the monitor initially invokes the function `validUname()` with argument `uname`; such a function is user-supplied (see input 2 above). If the function returns `false`, the monitor flags the violation and halts (line 10): this corresponds to the external assertion violation $\mathtt{no}_P^A$ in $[\![S^A_{auth}]\!]$. Otherwise, if `validUname()` returns `true`, the message is forwarded to the *server* (line 5). The function used to forward the message (`!!`), which is part of `lchannels`, corresponds to the external output operator ▲ of $[\![S^A_{auth}]\!]$; it returns a continuation channel that is stored in `cont`. To associate the payload identifiers of S^A_{auth} to their current values, the monitors maintain a mapping, called `payloads`. In this case, the value of `uname` is stored (line 7) since it is used later on in S_{auth}. Finally, the monitor moves to the next state `sendChoice1` (§ 5.2), passing the channel stored in `cont` to continue the protocol (line 8).

$$
\begin{aligned}
&[\![S^A_{res}]\!] = \\
&\quad ▼\{\mathtt{Succ}(tok : \mathsf{Str}).\mathsf{if}\ \mathsf{isB}_{\mathsf{Str}}(tok) \\
&\qquad \mathsf{then}\ \mathsf{if}\ validTok(tok, uname) \\
&\qquad\ \mathsf{then}\ \triangleleft\mathtt{Succ}(tok).[\![S_{succ}]\!] \\
&\qquad\ \mathsf{else}\ \mathtt{no}^A_E\ \mathsf{else}\ \mathtt{no}_E, \\
&\quad \mathtt{Fail}(code : \mathsf{Int}).\mathsf{if}\ \mathsf{isB}_{\mathsf{Int}}(code) \\
&\qquad \mathsf{then}\ \mathsf{if}\ \mathsf{tt}\ \mathsf{then}\ \triangleleft\mathtt{Fail}(code).Y \\
&\qquad \mathsf{else}\ \mathtt{no}^A_E\ \mathsf{else}\ \mathtt{no}_E\}
\end{aligned}
$$

```
1  def sendChoice1(srv:In[Choice1],Client:CM):Any = {
2    srv ? {
3      case msg @ Succ(_) =>
4        if (validTok(msg.tok, payloads.Auth.uname)) {
5          Client.send(msg)
6          /* Continue according to S_succ */
7        } else {
8          /* EXTERNAL VIOLATION (assertion) */
9        }
10     case msg @ Fail(_) =>
11       Client.send(msg)
12       receiveAuth(msg.cont, External)
13 } }
```

Figure 13 Comparison between the formal and implementation synthesis of the external choice.

According to S^A_{res}, the server can choose to send either `Succ` or `Fail`. The monitor waits to receive either of the options from the *server*, using the method `?` from `lchannels` (line 2). This corresponds to the external input operator of the monitor calculus (▼) used in $[\![S^A_{res}]\!]$, which can also receive both options from the *server*.

- If the *server* sends `Succ`(toc), the first case is selected (line 3). The monitor evaluates the assertion `validTok` on *tok* and *uname* (stored in § 5.2, and now retrieved from the `payloads` mapping). If it is satisfied, the message is forwarded to the client (line 5) via CM's `send` method, which corresponds to the internal send operator ($\triangleleft$) in the monitor calculus. The monitor then proceeds according to the monitor $[\![S_{succ}]\!]$. Otherwise, the monitor logs a violation and halts (line 8); similarly, $[\![S^A_{res}]\!]$ flags the violation $\mathtt{no}^A_E$ indicating an external assertion violation.
- Instead, if the *server* sends `Fail` (line 10), the monitor forwards it to the client; there are no assertion checks here, as the assertion after `Fail` in $[\![S^A_{res}]\!]$ is `tt`. Then, following the recursion in $[\![S^A_{res}]\!]$, the monitor (on line 12) loops to `receiveAuth` (§ 5.2).

Unlike the synthesised code of `receiveAuth` (that handles the previous external choice, in § 5.2), there is no catch-all case for unexpected messages from the *server*. This is by design. As explained above we use `lchannels` to interact with the "trusted" external side, hence the interaction with the server is typed, and a catch-all case would be unreachable code. Still, `lchannels` throws an exception (crashing the monitor) if it receives an invalid message – which corresponds to the monitor $[\![S^A_{res}]\!]$ flagging an external violation via rule [MEV].

6 Empirical Evaluation

We evaluate the feasibility of our implementation by measuring the overheads induced by the run-time checks of our synthesised monitors (§ 5). We consider 3 application protocols, modelled as session types, as our benchmarks:

1. A ping-pong protocol, based on a request-response over HTTP (a style of protocol that is typical, *e.g.,* in applications based on web services). Although it is a fairly simple protocol, our implementation uses HTTP to carry ping/pong messages, highlighting the fact that our generated monitors are independent from the message transport in use;
2. A fragment of the Simple Mail Transfer Protocol (SMTP) [47]. This benchmark represents a more complex protocol featuring nested internal/external choices;
3. A fragment of the HTTP protocol, also featuring sequences of nested internal/external choices.

Ping-pong over HTTP. In this protocol, a client is expected to recursively send messages with label `Ping` to the server which, in turn, replies with `Pong`. The protocol proceeds until the client sends `Quit`. The client-side protocol is shown below (the server-side is dual).

$$S_{pong} \;=\; \mathsf{rec}\ X.(\oplus\{!\texttt{Ping}().?\texttt{Pong}().X,\ !\texttt{Quit}()\})$$

Notice that the protocol has no explicit reference to HTTP. In fact, we use HTTP as a mere message transport, by providing a suitable Connection Manager to the synthesised monitor (which is transport-agnostic). Concretely, the ping-pong is implemented with the server handling requests on an URL like `http://127.0.0.1/ping`, and the client performing a `GET` request on that URL, and reading the response. For this benchmark, the setup is:

- the client is on the internal side of the generated monitor, hence subject to scrutiny;
- the server is on the external side of the generated monitor.

As untrusted client we use a standard, unmodified load testing tool: Apache JMeter (`https://jmeter.apache.org/`) configured to send HTTP requests at an increasing rate.

SMTP. We model a fragment of the SMTP protocol (server-side) as the session type S_{smtp}:

$$\begin{aligned}
S_{smtp} \;&=\; !\texttt{M220}(msg:\mathsf{Str}).\&\{?\texttt{Helo}(host:\mathsf{Str}).!\texttt{M250}(msg:\mathsf{Str}).S_{mail}, ?\texttt{Quit}().!\texttt{M221}(msg:\mathsf{Str})\} \\
S_{mail} \;&=\; \mathsf{rec}\ X.(\&\{?\texttt{MailFrom}(addr:\mathsf{Str}).!\texttt{M250}(msg:\mathsf{Str}).\mathsf{rec}\ Y.(\&\{ && (7)\\
&\qquad ?\texttt{RcptTo}(addr:\mathsf{Str}).!\texttt{M250}(msg:\mathsf{Str}).Y, && (8)\\
&\qquad ?\texttt{Data}().!\texttt{M354}(msg:\mathsf{Str}).?\texttt{Content}(txt:\mathsf{Str}).!\texttt{M250}(msg:\mathsf{Str}).X, && (9)\\
&\qquad ?\texttt{Quit}().!\texttt{M221}(msg:\mathsf{Str})\}), ?\texttt{Quit}().!\texttt{M221}(msg:\mathsf{Str})\})
\end{aligned}$$

When a client establishes a connection, the server sends a welcome message (`M220`), and waits for the client to identify itself (`Helo`). Then, the client can recursively send emails by specifying the sender and recipient address(es), followed by the mail contents. The client can send multiple emails by repeating the loop on "X" between lines (7) and (9).

The SMTP protocol runs over TCP/IP. The specification above (and the synthesised monitors) are again transport-agnostic: we handle TCP/IP sockets by providing a suitable Connection Manager to the synthesised monitor.

For this benchmark, the setup used is "dual" to that of the HTTP ping-pong benchmark above:

- the server is on the internal side of the generated monitor, hence subject to scrutiny;
- the client is on the external side of the generated monitor.

For this experiment, we implement an SMTP client that sends emails to the server, and measures the response time. We take such measurements against two (untrusted and monitored) servers, both configured to accept incoming emails and discard them:

1. a default instance of `smtpd` from the Python standard library;[1]
2. a default instance of Postfix,[2] one of the most used SMTP servers [59].

HTTP. In this benchmark, we do *not* use HTTP as a mere message transport (unlike the ping-pong benchmark above). Rather, we model HTTP headers, requests, and responses with a session type, which we use to synthesise a monitor that checks the interactions between a trusted server and an untrusted client. We focus on a fragment of HTTP that is sufficient for supporting typical client-server interactions (*e.g.,* when the client is the Mozilla Firefox browser). The HTTP session type (here omitted due space reasons) and its (trusted) server implementation are adapted from the `lchannels` examples [55]. For benchmarking, we use Apache JMeter (`https://jmeter.apache.org/`) as untrusted client.

Benchmarking setups and measurements. In all of our benchmarks, we study the overhead of our synthesised monitors by comparing:

- an *unsafe* setup: the client and server interact directly;
- a *monitored* setup: communication between the trusted and untrusted components is mediated by our synthesised monitors, which halts when it detects a violation – as described earlier in Fig. 11.

We follow a multi-faceted approach, as advocated by [8], and base our study on three measurements: average response time, average CPU utilisation, and maximum memory consumption. The response time is arguably the most important measurement, since slower response times can be immediately perceived when interacting with a monitored system. We measure them by running experiments of increasing length: for ping-pong and HTTP, we perform an increasing number of request-response loops, whereas for SMTP, we send an increasing number of emails. The general expectation is: for longer experiments, the average response time and CPU usage should decrease, while the maximum memory consumption should increase. We repeat each experiment 30 times, and we plot the average of all results.

In our benchmarks, overheads can have two forms:

Overhead 1: the translation and duplication of messages being forwarded between client and server;

Overhead 2: the run-time checks needed to ensure that the desired session type is being respected.

Overhead 1 is unavoidable for the most part. By their own nature, partial identity monitors (like ours) must receive and forward all messages. This overhead can only be minimised by using more efficient message transports. By contrast, overhead 2 is specifically caused by our monitor synthesis. Our benchmarks were specifically designed to accurately capture this latter form of overhead. In order to better distinguish overhead 1 from overhead 2, our benchmarks run the trusted side (client or server) and the synthesised monitors on a same JVM instance, where they interact in the most efficient way (*i.e.,* through the `LocalChannel` transport provided by the `lchannels` library). This minimises overhead 1,

[1] `https://docs.python.org/3/library/smtpd.html`
[2] `http://www.postfix.org/`

and allows us to better observe the impact of overhead 2. Clearly, the untrusted side of each benchmark (*i.e.,* the black-box client or server being monitored) always runs as a separate process.

Despite this, our synthesised monitors can still be deployed independently of the trusted side (*i.e.,* on their own JVM, possibly across a network) because they are agnostic to the message transports in use; this is made possible by the use of connection managers and `lchannels`. We demonstrate this capability by also taking measurements for a *detached* setup, where the trusted component and monitor run on separate JVMs (on a same host), and interact via TCP/IP (through a suitable message transport for `lchannels`). This setup is more flexible, but the slower message transport increases overhead 1. We implemented this setup for ping-pong and SMTP, measuring their response times.

Results and analysis. The benchmark results are reported Fig. 14. For the ping-pong benchmark (Fig. 14a), the impact of monitors is noticeable but limited: for the "monitored" setup (which highlights overhead 2), the response times are less than 14% slower; the "detached" monitor setup is unsurprisingly slower, due to its slower message transport (which increases overhead 1). For the SMTP benchmark (Figures 14b and 14c), we can observe different behaviours:

- the Python `smtpd` server (Fig. 14b) has extremely fast response times: it is essentially a dummy server that receives emails and does nothing with them. This is also evident from the CPU usage: it constantly increases, because the SMTP client receives immediate responses, no matter how many emails it sends, with or without a monitor. Consequently, our monitors cause a relatively high impact on such fast response times (almost 34%);
- the Postfix SMTP server (Fig. 14c) is more realistic: unlike Python `smtpd`, it takes some time (with fluctuations) to process each email and respond to the client. Consequently, our monitors have a relatively small impact on the response times (less than 7%).

As in the case of ping-pong, the "detached" monitor setup for both SMTP benchmarks is slower, as it uses a slower message transport (which increases overhead 1). Finally, the HTTP benchmark (Fig. 14d) shows a response time overhead that is below 5%. By and large, these overhead levels are tolerable for many applications that are not mission critical, and are comparable to the overhead experienced when running state-of-the-art RV tools [13].

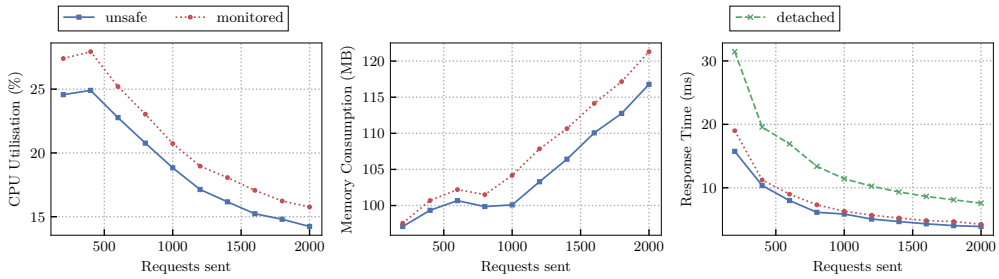
7 Conclusion

We presented a formal analysis for the *monitorability limits* of (binary) session types w.r.t. a partial-identity monitor model; to wit, this is the first monitorability assessment of session types. We couple this study with an implementation of session monitor synthesis.

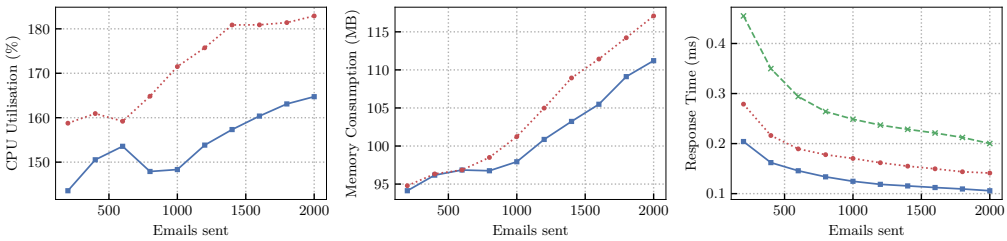
More in detail, our contributions are the following. On the the theoretical side, we provide the first treatment of the *monitorability* of session types, and *detection-soundness* and *detection-completeness* properties of session monitors, and we prove that our autogenerated session monitors enjoy both the former and (to a lesser extent) the latter. We also present an impossibility result of completeness for our black-box monitoring setup – which is a novel result to the area of session type monitoring. On the practical side, we evaluate the viability of our implementation (called `STMonitor`) via benchmarks. The results show that our monitor synthesis procedure only introduces limited overheads.

7.1 Related Work

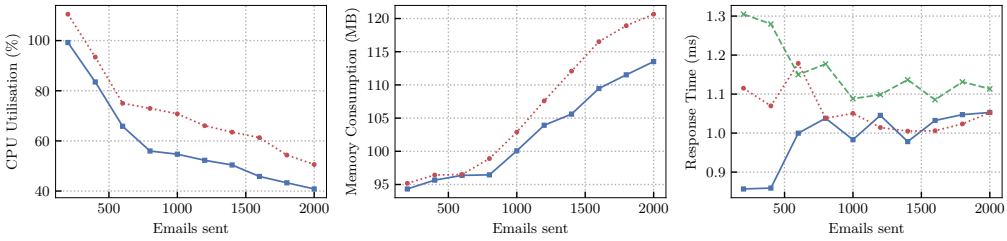
Several papers address the monitoring of session-types-based protocols – but no previous work studies the formal problem of session monitorability; furthermore, their approaches differ from ours in various ways, as we now discuss.



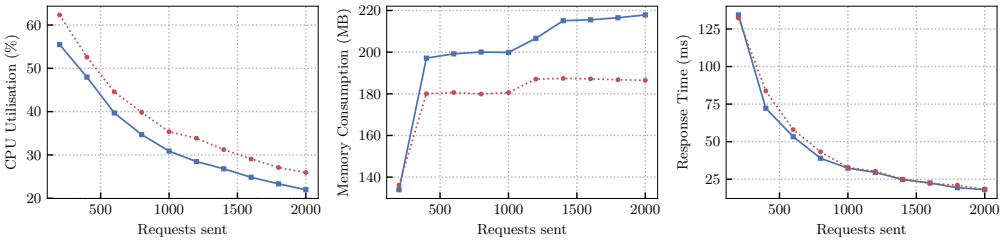
(a) Ping-pong over HTTP (trusted server, untrusted client). Monitored response time overhead: 13.82%.



(b) SMTP Python session (trusted client, untrusted server). Monitored response time overhead: 33.98%.



(c) SMTP Postfix session (trusted client, untrusted server). Monitored response time overhead: 6.68%.



(d) HTTP session (trusted server, untrusted client). Monitored response time overhead: 4.81%.

Figure 14 Benchmark results: average CPU usage, maximum memory consumption, and average response time (30 runs, 2 CPUs (Intel Pentium Gold G5400 @ 3.70GHz), 8 GB RAM, Ubuntu 20.04).

The work [17] formalises a theory of process networks including monitors generated from (multiparty) session types. The main differences with our work are:

1. the design of [17] is based on a global, centralised router providing a *safe transport network* that dispatches messages between participant processes; correspondingly, its implementation [24, 50, 40] includes a Python library for monitored processes to access the safe transport network. By contrast, we do not assume a specific message routing system, and our theory and implementation address the monitoring of black-box components;
2. the results in [17] do not consider limits related to session monitorability. Their results (*e.g., transparency*) are analogous to our *detection soundness* (Theorem 15), *i.e.,* synthesised monitors do not disrupt communications of well-typed processes; they do not address *completeness* (Theorem 19 and Theorem 21), *i.e.,* to what extent can a monitor detect ill-typed processes.

Furthermore, our work and [17] differ in a fundamental design choice: when our monitors detect an invalid message, they flag a violation and halt – whereas monitors in [17] drop invalid messages, and keep forwarding the rest. The latter is akin to *runtime enforcement via suppressions* [9]; studying this design with our theory is interesting future work.

Our protocol assertions (§ 5.1) are reminiscent of *interaction refinements* in [48], that are also statically generated (by an F# type provider), and dynamically enforced when messages are sent/received. However, our approach and design are different from [48]: we synthesise session monitoring processes that can be deployed over a network, to instrument black-box processes – whereas [48] expects the runtime-verified code to be written with a specific language and framework, and injects dynamic checks in the program executable. Furthermore, the work [48] does not address session monitorability limits.

The work [49] proposes a methodology to supervise (multiparty) session protocols, and recover them in case of failure of some component; it also includes an implementation in Erlang. Similarly to this work, in [49] each component is observed by a session monitor; unlike this work, [49] does not address any aspect of session monitorability, and focuses on proving that its recovery strategy does not deadlock.

The work by Gommerstadt *et al.* [34] considers a partial identity monitor model for session types that is close to the one discussed in § 3. They however do not provide any synthesis function and assume that monitors are constructed by hand. To complement this, they define a dedicated type system to prove that the monitor code behaves as a partial identity, *e.g.,* it forwards messages in the correct order, without dropping them. They do not study session monitorability. To our knowledge, their approach has not been implemented as a tool nor has it been assessed empirically either.

Melgratti and Padovani [46] propose monitors that act as wrappers around a session library. This technique effectively *inlines* the monitors in the monitored process code. In fact, their implementation assumes that the processes under scrutiny are written in OCaml using the FuSE library. In contrast, we synthesise *outline* monitors as independent processes that observe black-box implementations written in *any* language/library. The work proves a series of results that are akin to our notion of monitoring soundness, without addressing completeness.

In separate work, Waye *et al.* [63] monitor black-box services, focusing exclusively on *request-response* protocols. Unlike our session-type monitors, they do not support protocols with prescribed sequences of internal/external choices and recursion. In fact, their contracts are analogous to enhanced assertions on transmitted/received values (reminiscent of the assertion introduced in § 5.1). Although they provide soundness results for their monitoring framework, they do not consider any further monitorability issues.

The recent work [35] presents a runtime verification framework for communication protocols (based on multiparty session types) in Clojure. Unlike this work, [35] expects

monitored applications to be written in a specific language and framework – whereas we address the monitoring of black-box processes. Again, [35] does not study session monitorability.

7.2 Future Work

This work is our first step along a new line of research on the relative power of static versus run-time verification methods. In general terms, given a calculus C with a type system T and run-time monitoring system M, *monitoring soundness* tells us whether M is flagging "real" errors according to T. Dually, *monitoring completeness* tells us whether T is too restrictive w.r.t. M (*i.e.,* whether T is rejecting too many processes that M deems well-behaved). In this work, we demonstrate a rather tight connection between the chosen process calculus (C) and session type system (T), and our session monitors (M): our synthesised monitors are sound (Theorem 15), and most processes rejected by the type system behave incorrectly (Theorem 19). Our plan is to study more instances of C, T and M – both in theory, and in practice.

One avenue worth exploring is that of increasing the observational powers of the monitoring setup considered, in order to extend session monitorability. The work by Aceto *et al.* [2] is a systematic study that considers a variety of extensions to the traditional monitoring setup (consisting of one monitor observing events describing the computation effected by the process under scrutiny). The extensions considered include traces that report process termination and events that could not have been produced at different stages of the computation (*i.e.,* refusals [52]). They also consider monitoring setups where a process is monitored over multiple runs. In each case, they show the maximal properties that can be monitored for in a sound and complete manner, characterised a syntactic fragments of the modal μ-calculus. We intend to consider how any of the proposed extensions would affect our monitorability results and the extent to which they are implementable in practice. Other bodies of work take a slightly different approach to monitorability, by weakening the completeness requirement from their notion of adequate monitoring [5, 6]. It would be worthwhile exploring the effect of having such weakened completeness requirements on the monitorability of session types.

Although we have limited ourselves to binary session, we plan to extend the framework above to the static and run-time verification of multiparty and asynchronous sessions [38, 39]. This will most likely require us to consider communicating monitors, that cooperate to aggregate observations made from analysing communications on distinct channels. For multiparty sessions, we can benefit from previous work [54, 55] where `lchannels` is used to implement multiparty protocols written in Scribble [58, 64]. Our implementations should also benefit from insights gained from numerous work on decentralised runtime verification [15, 12, 25]. For both multiparty and asynchronous sessions, we can benefit from the research on precise session subtyping [33, 32, 22].

In this work, our session monitors adhere to the *"fail-fast"* design methodology: if a protocol violation occurs, the monitor flags the violation and halts. In the practice of distributed systems, "fail-fast" is advocated as an alternative to defensive programming [20]; it is also in line with existing literature on runtime verification [13]. As mentioned above, an interesting research direction is to adapt our session monitorability framework to *suppressions* [9], *i.e.,* by dropping invalid messages without halting the monitor, as in [17].

Finally, we plan to investigate how to handle violations by adding *compensations* to our session types – *i.e.,* by formalising how the protocol should proceed if a violation is detected at a certain stage. In this setting, the monitors would play a more active role in handling violations, and their synthesis would need to be more sophisticated; this new research could be related to the work on session recovery [49].

References

1 Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. Monitoring for silent actions. In *FSTTCS*, volume 93 of *LIPIcs*, pages 7:1–7:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

2 Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. A framework for parameterized monitorability. In *FoSSaCS*, volume 10803 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 2018.

3 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Adventures in monitorability: from branching to linear time and back again. *Proc. ACM Program. Lang.*, 3(POPL):52:1–52:29, 2019. doi:10.1145/3290365.

4 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. An operational guide to monitorability. In *SEFM*, volume 11724 of *Lecture Notes in Computer Science*, pages 433–453. Springer, 2019. doi:10.1007/978-3-030-30446-1_23.

5 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. The best a monitor can do. In *CSL*, volume 183 of *LIPIcs*, pages 7:1–7:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

6 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. An operational guide to monitorability with applications to regular properties. *Softw. Syst. Model.*, 20(2):335–361, 2021. doi:10.1007/s10270-020-00860-z.

7 Luca Aceto, Duncan Paul Attard, Adrian Francalanza, and Anna Ingólfsdóttir. A choreographed outline instrumentation algorithm for asynchronous components. *CoRR*, abs/2104.09433, 2021. URL: https://arxiv.org/abs/2104.09433.

8 Luca Aceto, Duncan Paul Attard, Adrian Francalanza, and Anna Ingólfsdóttir. On benchmarking for concurrent runtime verification. In *FASE*, volume 12649 of *Lecture Notes in Computer Science*, pages 3–23. Springer, 2021.

9 Luca Aceto, Ian Cassar, Adrian Francalanza, and Anna Ingólfsdóttir. On runtime enforcement via suppressions. In *CONCUR*, volume 118 of *LIPIcs*, pages 34:1–34:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.CONCUR.2018.34.

10 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *POPL*, pages 201–214. ACM, 2011. doi:10.1145/1926385.1926409.

11 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Found. Trends Program. Lang.*, 3(2-3):95–230, 2016. doi:10.1561/2500000031.

12 Duncan Paul Attard and Adrian Francalanza. Trace partitioning and local monitoring for asynchronous components. In *SEFM*, volume 10469 of *Lecture Notes in Computer Science*, pages 219–235. Springer, 2017.

13 Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In *Lectures on Runtime Verification*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2018. doi:10.1007/978-3-319-75632-5_1.

14 David A. Basin, Thibault Dardinier, Lukas Heimes, Srdan Krstic, Martin Raszyk, Joshua Schneider, and Dmitriy Traytel. A formally verified, optimized monitor for metric first-order dynamic logic. In *IJCAR (1)*, volume 12166 of *Lecture Notes in Computer Science*, pages 432–453. Springer, 2020. doi:10.1007/978-3-030-51074-9_25.

15 Andreas Bauer and Yliès Falcone. Decentralised LTL monitoring. *Formal Methods Syst. Des.*, 48(1-2):46–93, 2016.

16 Jeremy Blackburn, Ivory Hernandez, Jay Ligatti, and Michael Nachtigal. Completely subtyping iso-recursive types. Technical Report CSE-071012, University of South Florida, 2012.

17 Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. *Theor. Comput. Sci.*, 669:33–58, 2017. doi:10.1016/j.tcs.2017.02.009.

18 Alan Brown, Jerry Fishenden, and Mark Thompson. *API Economy, Ecosystems and Engagement Models*, pages 225–236. Palgrave Macmillan UK, London, 2014. doi:10.1057/9781137443649_13.

19 Christian Batrolo Burlò, Adrian Francalanza, and Alceste Scalas. On the monitorability of session types, in theory and practice (extended version), 2021. arXiv:2105.06291.

20 Francesco Cesarini and Simon Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009.

21 Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC*, volume 7173 of *Lecture Notes in Computer Science*, pages 25–45. Springer, 2011. doi:10.1007/978-3-642-30065-3_2.

22 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the preciseness of subtyping in session types. *Log. Methods Comput. Sci.*, 13(2), 2017. doi:10.23638/LMCS-13(2:12)2017.

23 Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *IEEE Symposium on Security and Privacy*, pages 813–830. IEEE Computer Society, 2015. doi:10.1109/SP.2015.55.

24 Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. *Formal Methods Syst. Des.*, 46(3):197–225, 2015. doi:10.1007/s10703-014-0218-8.

25 Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring hyperproperties. *Formal Methods Syst. Des.*, 54(3):336–363, 2019.

26 Adrian Francalanza. A theory of monitors - (extended abstract). In *FoSSaCS*, volume 9634 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2016. doi:10.1007/978-3-662-49630-5_9.

27 Adrian Francalanza. Consistently-detecting monitors. In *CONCUR*, volume 85 of *LIPIcs*, pages 8:1–8:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CONCUR.2017.8.

28 Adrian Francalanza. A Theory of Monitors. *Information and Computation*, page 104704, 2021. doi:10.1016/j.ic.2021.104704.

29 Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. A foundation for runtime monitoring. In *RV*, volume 10548 of *Lecture Notes in Computer Science*, pages 8–29. Springer, 2017. doi:10.1007/978-3-319-67531-2_2.

30 Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. Monitorability for the hennessy-milner logic with recursion. *Formal Methods Syst. Des.*, 51(1):87–116, 2017. doi:10.1007/s10703-017-0273-z.

31 Adrian Francalanza and Jasmine Xuereb. On implementing symbolic controllability. In *COORDINATION*, volume 12134 of *Lecture Notes in Computer Science*, pages 350–369. Springer, 2020. doi:10.1007/978-3-030-50029-0_22.

32 Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *J. Log. Algebraic Methods Program.*, 104:127–173, 2019. doi:10.1016/j.jlamp.2018.12.002.

33 Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for asynchronous multiparty sessions. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021. doi:10.1145/3434297.

34 Hannah Gommerstadt, Limin Jia, and Frank Pfenning. Session-typed concurrent contracts. In *ESOP*, volume 10801 of *Lecture Notes in Computer Science*, pages 771–798. Springer, 2018. doi:10.1007/978-3-319-89884-1_27.

35 Ruben Hamers and Sung-Shik Jongmans. Discourje: Runtime verification of communication protocols in clojure. In *TACAS (1)*, volume 12078 of *Lecture Notes in Computer Science*, pages 266–284. Springer, 2020. doi:10.1007/978-3-030-45190-5_15.

36 Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

37 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

38 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.

39 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.

40 Raymond Hu, Rumyana Neykova, Nobuko Yoshida, Romain Demangeon, and Kohei Honda. Practical interruptible conversations - distributed dynamic verification with session types and python. In *RV*, volume 8174 of *Lecture Notes in Computer Science*, pages 130–148. Springer, 2013. doi:10.1007/978-3-642-40787-1_8.

41 Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *Proc. ACM Program. Lang.*, 1(ICFP):38:1–38:28, 2017. doi:10.1145/3110282.

42 Limin Jia, Hannah Gommerstadt, and Frank Pfenning. Monitors and blame assignment for higher-order session types. In *POPL*, pages 582–594. ACM, 2016. doi:10.1145/2837614.2837662.

43 Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, 4(1-2):2–16, 2005. doi:10.1007/s10207-004-0046-8.

44 Jay Ligatti, Jeremy Blackburn, and Michael Nachtigal. On subtyping-relation completeness, with an application to iso-recursive types. *ACM Trans. Program. Lang. Syst.*, 39(1):4:1–4:36, 2017. doi:10.1145/2994596.

45 Kim Guldstrand Larsen Luca Aceto, Anna Ingólfsdóttir and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.

46 Hernán C. Melgratti and Luca Padovani. Chaperone contracts for higher-order sessions. *Proc. ACM Program. Lang.*, 1(ICFP):35:1–35:29, 2017. doi:10.1145/3110279.

47 Network Working Group. RFC 5321: Simple Mail Transfer Protocol. https://tools.ietf.org/html/rfc5321, 2008.

48 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time API generation of distributed protocols with refinements in f#. In *CC*, pages 128–138. ACM, 2018. doi:10.1145/3178372.3179495.

49 Rumyana Neykova and Nobuko Yoshida. Let it recover: multiparty protocol-induced recovery. In *CC*, pages 98–108. ACM, 2017. doi:10.1145/3033019.

50 Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. SPY: local verification of global protocols. In *RV*, volume 8174 of *Lecture Notes in Computer Science*, pages 358–363. Springer, 2013. doi:10.1007/978-3-642-40787-1_25.

51 Doron A. Peled. Specification and verification using message sequence charts. *Electron. Notes Theor. Comput. Sci.*, 65(7):51–64, 2002. doi:10.1016/S1571-0661(04)80484-5.

52 Iain Phillips. Refusal testing. *Theor. Comput. Sci.*, 50:241–284, 1987.

53 Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

54 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In *ECOOP*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ECOOP.2017.24.

55 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming (artifact). *Dagstuhl Artifacts Ser.*, 3(2):03:1–03:2, 2017. doi:10.4230/DARTS.3.2.3.

56 Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In *ECOOP*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.21.

57 Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000. doi:10.1145/353323.353382.

58 Scribble homepage, 2020. URL: http://www.scribble.org.

59 SecuritySpace. Mail (MX) server survey, 2021. URL: http://www.securityspace.com/s_survey/data/man.202103/mxsurvey.html.

60 Paula Severi and Mariangiola Dezani-Ciancaglini. Observational equivalence for multiparty sessions. *Fundam. Informaticae*, 170(1-3):267–305, 2019. doi:10.3233/FI-2019-1863.

61 Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *ISSTA*, pages 174–184. ACM, 2007. doi:10.1145/1273463.1273487.

62 Fu Song and Tayssir Touili. Model-checking software library API usage rules. In *IFM*, volume 7940 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2013. doi:10.1007/978-3-642-38613-8_14.

63 Lucas Waye, Stephen Chong, and Christos Dimoulas. Whip: higher-order contracts for modern services. *Proc. ACM Program. Lang.*, 1(ICFP):36:1–36:28, 2017.

64 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble protocol language. In *TGC*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013. doi:10.1007/978-3-319-05119-2_3.

λ-Based Object-Oriented Programming

Marco Servetto
ECS, Victoria University of Wellington, New Zealand

Elena Zucca
DIBRIS, University of Genova, Italy

Abstract

We show that a minimal subset of Java 8 excluding classes supports a simple and natural programming style, which we call *λ-based object-oriented programming*. That is, on one hand the programmer can use *tuples* in place of objects (class instances), and tuples can be desugared to lambdas following their classical encoding in the λ-calculus. On the other hand, lambdas can be equipped with additional behaviour, thanks to the fact that they may implement interfaces with default methods, hence inheritance and dynamic dispatch are still supported. We formally describe the encoding by a translation from FJ_λ, an FJ variant including lambdas and interfaces with default methods, to FJ_λ^-, a subset of FJ_λ with no classes (hence no constructors and fields). We provide several examples illustrating this novel programming style.

2012 ACM Subject Classification Software and its engineering → Object oriented languages; Software and its engineering → Functional languages

Keywords and phrases Programming paradigms, Java, lambda-calculus

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.21

Category Pearl

Acknowledgements We warmly thank the anonymous referees for suggestions which greatly improved the paper.

1 Introduction

Java 8 introduced lambdas and default interface methods. In Java 8, an interface with exactly one abstract method can be instantiated with the convenient lambda syntax. Lambdas are intended to represent first class functions; indeed, when such abstract method is invoked on a lambda, the body is executed as in standard application (β-rule). However, thanks to default methods, also interfaces with multiple methods can be instantiated with lambdas. In this way, lambdas also behave as regular objects, making it possibile to write object-oriented code without any need of classes and constructor invocations. Consider the following Java code example:

```
interface Person{
  String name();
  default String greet(){
    return "Hi, I'm "+this.name()+"; nice to meet you!";
  }
}
Person bob = ()->"bob";
bob.greet();

interface GamerPerson extends Person{
  default String greet(){
    return "Hi, I'm "+this.name()+"; and I love computer games!";
  }
}
Person p = (GamerPerson)()->"charles";
p.greet();// dynamic dispatch
```

In this example, the lambda, rather than be used as a first-class function, serves a role similar to an object with a `name` field. Correspondingly, the unique abstract method `String name();` behaves as a getter, even though a field `name` is never really declared.

At first, this programming style may look an odd curiosity or even a form of misguided code obfuscation. We started playing with these kinds of programming patterns as a funny exercise, to see how far we could push this; unexpectedly, in the end we realized that it is possible to program in a pretty natural way in such a paradigm, hence in a small Java 8 subset excluding two of the most iconic Java keywords: `class` and `new`. We call this programming style *λ-based object-oriented programming*: it can be seen as a novel way to conciliate OO and functional programming, that neither simply encodes one approach into the other, nor just merges constructs from both paradigms.

In other words, our aim is to explore a programming style which is basically *functional* (first-class values are only lambdas) but where functions may have, besides application, additional behaviour, thus supporting inheritance and dynamic binding, hence code reuse, as in object-oriented programming. Moreover, our aim is to present (and encode) this approach by a minimal and clean calculus. In other words, our focus here is not on increasing expressive power, but on simplicity: encoding the same features with fewer constructs.

To illustrate λ-based object-oriented programming, first of all in Section 2 we present a core language supporting this paradigm, by means of a simple calculus $\mathrm{FJ}^{-}_{\lambda}$, in the style of Featherweight Java (FJ) [8]. An $\mathrm{FJ}^{-}_{\lambda}$ program is a table of interfaces, and expressions are only variables, method calls and lambdas. In a sense, this expression language is lambda-calculus enriched by the foundational feature of object-oriented programming, that is, dynamically dispatched method call. While some methods are implemented with lambdas, most code is provided inside (default) method bodies keeping the natural code organization typical of OO programming.

Then, we enrich $\mathrm{FJ}^{-}_{\lambda}$ by classes, fields and constructors, obtaining an extended calculus FJ_{λ}, and in Section 3 we show a simple translation from FJ_{λ} to $\mathrm{FJ}^{-}_{\lambda}$, thus formally proving that such constructs are redundant language features. The translation is inspired by the classical encoding of tuples in the lambda-calculus; however, some additional work is needed to use the encoding in a language with only nominal types (interface names). We show that the translation preserves typing and semantics.

In Section 4, we provide several programming examples illustrating how to rely on this paradigm in a real language, notably a Java 8 subset excluding classes. Finally, in Section 5 we discuss possibile extensions and relation with other works, and in Section 6 we summarize the contribution of the paper and outline future directions.

2 The $\mathrm{FJ}^{-}_{\lambda}$ and FJ_{λ} calculi

Syntax, reduction rules, and typing rules of $\mathrm{FJ}^{-}_{\lambda}$ are given in Figure 1. We write Ds as metavariable for $D_1 \dots D_n$, $n \geq 0$, and analogously for other sequences.

As anticipated, an $\mathrm{FJ}^{-}_{\lambda}$ program is a table of interfaces, and expressions are only variables, method calls and λ-expressions (lambdas). In Java, λ-abstractions can only be typed when occurring in a context requiring a given type (called the *target type*). Here, we directly assume lambdas to be annotated with their interface type, since the issue of deriving this annotation from the context is orthogonal to our topic, and has been faced in other works [2, 6]. Moreover, in the calculus we use the lambda-calculus syntax, both to help readability and to stress that it can be seen both as a subset of Java, and, conversely, as a λ-calculus equipped with an interface table.

P	::=	Ds	program
D	::=	$\texttt{interface}\ I\ \texttt{extends}\ Is\ \{\ IMs\ \}$	declaration
IM	::=	$MH\texttt{;} \mid \texttt{default}\ M$	interface method
MH	::=	$T\ m\texttt{(}T_1\ x_1\texttt{,} \ldots\texttt{,}\ T_n\ x_n\texttt{)}$	method header
M	::=	$MH\ \{\texttt{return}\ e\texttt{;}\}$	method
R, S, T	::=	I	type
e	::=	$x \mid e\texttt{.}m\texttt{(}es\texttt{)} \mid (\lambda xs.e)^I$	expression
v	::=	$(\lambda xs.e)^I$	value
$\mathcal{E}$	::=	$[\,]\texttt{.}m\texttt{(}es\texttt{)} \mid v\texttt{.}m\texttt{(}vs\ [\,]\ es\texttt{)}$	evaluation context
Γ	::=	$x_1\texttt{:}T_1 \ldots x_n\texttt{:}T_n$	typing context

$$(\text{CTX})\ \frac{e \to_P e'}{\mathcal{E}[e] \to_P \mathcal{E}[e']} \qquad (\text{INVK})\ \frac{}{v\texttt{.}m\texttt{(}vs\texttt{)} \to_P e[xs \leftarrow vs][\texttt{this} \leftarrow v]}\ \begin{array}{l}\mathsf{typeof}(v) = T \\ \mathsf{mbody}_P(T, m) = \langle xs,\ e\rangle\end{array}$$

$$(\beta)\ \frac{}{v\texttt{.}m\texttt{(}vs\texttt{)} \to_P e[xs \leftarrow vs][\texttt{this} \leftarrow v]}\ \begin{array}{l}v = (\lambda xs.e)^I \\ !\mathsf{absmeth}_P(I) = m\end{array}$$

$$(\text{T-PROG})\ \frac{\vdash_P D_1 \ldots \vdash_P D_n}{\vdash P}\ P = D_1 \ldots D_n$$

$$(\text{T-INTERFACE})\ \frac{I \vdash_P IM_1 \ldots I \vdash_P IM_n}{\vdash_P \texttt{interface}\ I\ \texttt{extends}\ Is\ \{\ IM_1 \ldots IM_n\ \}}\ Is \subseteq \mathsf{inames}(P)$$

$$(\text{T-MH})\ \frac{}{T \vdash_P R\ m\texttt{(}T_1\ x_1\texttt{,} \ldots\texttt{,}\ T_n\ x_n\texttt{)}}\ \begin{array}{l}\{R, T_1, \ldots, T_n\} \subseteq \mathsf{tnames}(P) \\ (T \leq_P S \text{ and } \mathsf{mtype}_P(S, m) = Ts \to R') \text{ imply} \\ \quad Ts = T_1 \ldots T_n \\ \quad R' = R\end{array}$$

$$(\text{T-ABS})\ \frac{T \vdash_P MH}{T \vdash_P MH\texttt{;}} \qquad (\text{T-DEFAULT})\ \frac{T \vdash_P M}{T \vdash_P \texttt{default}\ M}$$

$$(\text{T-METH})\ \frac{\vdash_P MH \quad \Gamma \vdash_P e : R'}{T \vdash_P MH\ \{\texttt{return}\ e\texttt{;}\}}\ \begin{array}{l}MH = R\ m\texttt{(}T_1\ x_1\texttt{,} \ldots\texttt{,}\ T_n\ x_n\texttt{)} \\ \Gamma = x_1\texttt{:}T_1 \ldots x_n\texttt{:}T_n\ \texttt{this:}T \\ R' \leq_P R\end{array}$$

$$(\text{T-VAR})\ \frac{}{\Gamma \vdash_P x : T}\ \Gamma(x) = T$$

$$(\text{T-INVK})\ \frac{\Gamma \vdash_P e_i : T_i \quad \forall i \in 0..n}{\Gamma \vdash_P e_0\texttt{.}m\texttt{(}e_1\texttt{,} \ldots\texttt{,}\ e_n\texttt{)} : R}\ \begin{array}{l}\mathsf{mtype}_P(T_0, m) = S_1 \ldots S_n \to R \\ T_i \leq_P S_i \quad \forall i \in 1..n\end{array}$$

$$(\text{T-LAM})\ \frac{\Gamma[x_1\texttt{:}T_1 \ldots x_n\texttt{:}T_n] \vdash_P e : S}{\Gamma \vdash_P (\lambda x_1 \ldots x_n.e)^I : I}\ \begin{array}{l}!\mathsf{absmtype}_P(I) = T_1 \ldots T_n \to R \\ S \leq_P R\end{array}$$

Figure 1 Formal definition of FJ^-_λ.

The formal definition is straightforward. Given a program P, $\mathsf{inames}(P)$ and $\mathsf{tnames}(P)$ are the declared interface names and type names; $\mathsf{typeof}(v)$ the (dynamic) type of value v, which for a lambda is its annotation; $\mathsf{mtype}_P(T, m)$ and $\mathsf{mbody}_P(T, m)$ the type and body of method m in T, if any; $!\mathsf{absmeth}_P(I)$ and $!\mathsf{absmtype}_P(I)$, only defined if I has exactly one abstract method[1], the name and type of such method; finally, $\leq_P$ the reflexive and transitive closure of the `extends` relation. For brevity, we omit the straightforward formal definitions. Only note that we require mtype and mbody, defined as in FJ, to be actually functions; this constraint implies that an interface I cannot inherit the same method m with different signatures, and cannot inherit more than one default method m, unless m is declared by I as well.

We denote by $\mathcal{E}[e]$ the expression obtained by filling the hole of the context $\mathcal{E}$ with e, and by $e[xs \leftarrow vs]$ the expression obtained from e by replacing variables xs with values vs. Typing contexts are assumed to represent finite maps from variables into types, hence the notation $\Gamma(x)$ is well-defined; we denote by $\Gamma[\Gamma']$ the type context which coincides with Γ' when the latter is defined, with Γ otherwise.

In Figure 2, we extend FJ^-_λ with classes, fields and constructor invocations. New productions are emphasized, and we only write the new reduction and typing rules. The extended calculus is similar to other calculi extending FJ with interfaces and lambdas [2, 6]. Here, we do not include subclassing, since our focus is to show that the role of classes as object's generators can be replaced by instantiating functional interfaces with lambdas; instead, we expect the role of subclassing for code reuse to be achieved by interface inheritance and default methods.

The reduction relation $\rightarrow_P$ is defined on closed expressions. The calculus enjoys standard properties; notably, reduction is deterministic, and the type system is sound, that is, reduction of (closed) well-typed expressions with respect to well-typed programs does not go stuck, as formally stated below. We write $\rightarrow^\star_P$ for the transitive and reflexive closure of $\rightarrow_P$.

▶ **Theorem 1** (Determinism). *If $e \rightarrow_P e_1$ and $e \rightarrow_P e_2$, then $e_1 = e_2$.*

Proof. By structural induction on e, observing that at most one (instantation of meta-)rule is applicable. ◀

▶ **Theorem 2** (Soundness). *If $\vdash P$, and $\emptyset \vdash_P e : T$, and $e \rightarrow^\star_P e'$, then either e' is a value or $e' \rightarrow_P e''$ for some e''.*

Proof. Straightforward adaptation of the proof provided for a richer language in [2]. ◀

Set $v^\infty ::= v \mid \infty$. The relation $e \Rightarrow_P v^\infty$, associating to an expression e its *semantics* in P, is defined as follows:

- $e \Rightarrow_P v$ if $e \rightarrow^\star_P v$
- $e \Rightarrow_P \infty$ if e has an infinite reduction sequence in P.

If $\vdash P$, and $\emptyset \vdash_P e : T$, then Theorem 1 and Theorem 2 above ensure that the semantics of e in P is well-defined, that is, $e \Rightarrow_P v^\infty$ for a unique v^∞.

[1] That is, is a *functional* interface.

P	$::=$	Ds	program
D	$::=$	$\texttt{interface}\ I\ \texttt{extends}\ Is\ \{\ IMs\ \}$	declaration
		$\mid \texttt{class}\ C\ \texttt{implements}\ Is\ \{Fs\ Ms\}$	
IM	$::=$	$MH \mid \texttt{default}\ M$	interface method
MH	$::=$	$T\ m\ (T_1\ x_1, \ldots, T_n\ x_n)$	method header
M	$::=$	$MH\ \{\texttt{return}\ e;\}$	method
F	$::=$	$T\ f;$	field
T	$::=$	$I \mid C$	type
e	$::=$	$x \mid e.m(es) \mid (\lambda xs.e)^I \mid \texttt{new}\ C(es) \mid e.f$	expression
v	$::=$	$(\lambda xs.e)^I \mid \texttt{new}\ C(vs)$	value
$\mathcal{E}$	$::=$	$[\,].m(es) \mid v.m(vs\ [\,]\ es) \mid \texttt{new}\ C(vs\ [\,]\ es) \mid [\,].f$	evaluation context
Γ	$::=$	$x_1{:}T_1 \ldots x_n{:}T_n$	typing context

$$(\text{FIELD})\ \frac{}{\texttt{new}\ C(v_1 \ldots v_n).f_i \rightarrow_P v_i}\quad \begin{array}{l}\mathsf{fields}_P(C) = T_1\ f_1; \ldots T_n\ f_n; \\ i \in 1..n\end{array}$$

$$(\text{T-CLASS})\ \frac{I \vdash_P M_1 \ldots I \vdash_P M_n}{\vdash_P \texttt{class}\ C\ \texttt{implements}\ Is\ \{T_1\ f_1; \ldots T_k\ f_k;\ \ M_1 \ldots M_n\}}\quad \begin{array}{l}T_1 \ldots T_k \subseteq \mathsf{tnames}(P) \\ I_s \subseteq \mathsf{inames}(P)\end{array}$$

$$(\text{T-FIELD})\ \frac{\Gamma \vdash_P e : C}{\Gamma \vdash_P e.f_i : T_i}\quad \begin{array}{l}\mathsf{fields}_P(C) = T_1\ f_1; \ldots T_n\ f_n; \\ i \in 1..n\end{array}$$

$$(\text{T-NEW})\ \frac{\Gamma \vdash_P e_i : T_i\ \ \forall i \in 1..n}{\Gamma \vdash_P \texttt{new}\ C(e_1, \ldots, e_n) : C}\quad \begin{array}{l}\mathsf{fields}_P(C) = S_1\ f_1; \ldots S_n\ f_n; \\ T_i \leq_P S_i\ \ \forall i \in 1..n\end{array}$$

Figure 2 Formal definition of FJ_λ.

3 Translation

First we explain the translation on a simple example: a class `Pair` with two fields, where for simplicity types `A` and `B` can be thought to be primitive types, e.g., `int` and `boolean`, respectively.

```
class Pair {
  A fst;
  B snd;
}
```

An instance of a class with n fields is essentially a tuple with n components, so our translation is based on the classical encoding of tuples in λ-calculus: a tuple is a function which, taken a *selector*, returns the corresponding component. In the example, a pair, e.g., $\texttt{mypair} = \langle 1, \texttt{true}\rangle$, is encoded by the function $\lambda \texttt{s}.\texttt{s}\,1\,\texttt{true}$, and there are only two expected selectors: $\texttt{fst} = \lambda x.\lambda y.x$ and $\texttt{snd} = \lambda x.\lambda y.y$. For instance, $\texttt{mypair}\,\texttt{fst}$ reduces to 1.

We consider now the problem of assigning types to the functions encoding tuples and selectors. Of course this is easy if we have polymorphic types. However, in each concrete case, there are only n different types of selectors which can be given as argument to the tuple, and for each of them a different result type. In other words, union types are enough. In a language with algebraic types, such as, e.g., Haskell, the same effect can be achieved by constructors which act as embeddings, as shown below.

```
type A = Int
type B = Bool
data AorB = FromA A | FromB B

type Fst = A -> B -> A
type Snd = A -> B -> B
data Sel = FromFst Fst | FromSnd Snd

type Pair = Sel -> AorB
--mypair = <1,True>
mypair :: Pair
mypair (FromFst s) = FromA (s 1 True)
mypair (FromSnd s) = FromB (s 1 True)

getFst :: Pair -> A
getFst p =
  let FromA a = p (FromFst (\a b -> a))
  in a

getSnd :: Pair -> B
getSnd p =
  let FromB b = p (FromSnd (\a b -> b))
  in b
```

Note that the getter methods could in principle raise a pattern matching error, but this will never happen at runtime.

The encoding in Java is based on the same idea. However, in this case the union of the result types is encoded by an interface, and the embedding of an element of type `A` into `AorB` is the constant function `(FromA)()->a`. On the other hand, it is enough to have the interface `Sel` corresponding to the various selectors, since in this case the embedding is silently obtained by subtyping.

```
interface AorB{
  default A toA(){/*error*/}
  default B toB(){/*error*/}
}

interface FromA extends AorB{ A toA();}
interface FromB extends AorB{ B toB(); }

interface Sel { AorB apply(A a, B b);}}

interface Pair {
  default A getFst(){
    return this.apply((Sel)(a,b)->(FromA)()->a).toA();}
  default B getSnd(){
    return this.apply((Sel)(a,b)->(FromB)()->b).toB();}
  AorB apply (Sel sel);
  }
```

For instance, the object `new Pair(myA,myB)` is encoded by `(Pair)s->s.apply(myA,myB)`. Also in this case the getter methods could in principle raise an error, but this will never happen at runtime. Hence, the body of the default methods `toA` and `toB` could be arbitrary well-typed expressions, which will be never executed, as indicated by the `/*error*/` comment. In the minimal syntax of $\mathrm{FJ}^{-}_{\lambda}$, such arbitrary expressions could be the recursive calls `this.toA()` and `this.toB()`. In full Java, an exception could be thrown.

This pattern is applied for all classes with $n \geq 2$ fields, as formally defined below; for each such class $n + 3$ interfaces are generated. Classes with zero or one field have specific (simpler) encodings, explained below.

As shown in the `Person` example before, a class with a single field can be encoded by an interface which defines a single abstract no-args getter method for such field, and an instance of the class can be encoded by a constant function which returns the field value.

A class with no fields, in a functional setting, has only one instance, and offers a set of methods to be invoked on such unique instance, which hence can be seen as class methods.The corresponding interface offers such methods, but also needs an abstract method, since the unique instance of the class should be encoded by a lambda. Note that an arbitrary abstract method, and an arbitrary lambda providing the implementation could be used, since the method will never be called. We conventionally use a method `dummy` with argument and return type `Void`, where `Void` is an empty interface, and as dummy lambda the identity function. In the following examples in full Java, we use a `void dummy()` method and the empty block as body of the dummy lambda.

We provide now the formal translation, denoted $[\![\_]\!]$. To the aim of this translation, we assume that, in the FJ_{λ} program to be translated, arguments of constructor invocations are only variable or values, since otherwise (possibly non terminating or stuck) reduction of arguments would be not simulated in the translation. This is not a restriction, since general constructor invocations can be encoded by auxiliary methods in FJ_{λ}, and could be translated adding local variable declarations in $\mathrm{FJ}^{-}_{\lambda}$. Moreover, as mentioned above, we assume a declaration `interface Void {}`. Finally, we assume that all the interface and method names introduced by the translation are fresh, that is, they do not clash with existing names. We first provide the translation of class declarations. The C at the beginning of interface names is necessary to get unique names. Moreover, the `to` methods are decorated by indexes, rather than field types as in the introductory example, since two fields could have the same type.

$[\![\texttt{interface}\ I\ \texttt{extends}\ Is\ \{IM_1 \dots IM_n\}]\!] = \texttt{interface}\ I\ \texttt{extends}\ Is\ \{[\![IM_1]\!] \dots [\![IM_n]\!]\}$

$[\![\texttt{class}\ C\ \texttt{implements}\ Is\ \{MH_1\ \{\texttt{return}\ e_1\texttt{;}\} \dots MH_n\ \{\texttt{return}\ e_n\texttt{;}\}\}]\!]$ = (zero fields)

```
interface C extends Is {
    default MH_1 {return [[e_1]];}
    ...
    default MH_n {return [[e_n]];}
    Void dummy(Void x);
}
```

$[\![\texttt{class}\ C\ \texttt{implements}\ Is\ \{T\ f\texttt{;}\ MH_1\ \{\texttt{return}\ e_1\texttt{;}\} \dots MH_n\ \{\texttt{return}\ e_n\texttt{;}\}\}]\!]$ = (one field)

```
interface C extends Is {
    default MH_1 {return [[e_1]];}
    ...
    default MH_n {return [[e_n]];}
    T getf();
}
```

$[\![\texttt{class}\ C\ \texttt{implements}\ Is\ \{T_1\ f_1\texttt{;} \dots T_k\ f_k\texttt{;}\ MH_1\ \{\texttt{return}\ e_1\texttt{;}\} \dots MH_n\ \{\texttt{return}\ e_n\texttt{;}\}\}]\!]$ = (≥ 2 fields)

```
interface CUnion {
    default T_1 to1(){return /*error*/;}
    ...
    default T_k tok(){return /*error*/;}
interface CFrom1 extends CUnion{to1();}
...
interface CFromk extends CUnion{tok();}
interface CSel{CUnion apply(T_1 x_1, ..., T_k x_k);}
interface C extends Is {
    default MH_1 {return [[e_1]];}
    ...
    default MH_n {return [[e_n]];}
    default T_1 getf_1(){return this.apply((λx_1 ... x_k.(λ.x_1)^CFromk)^CSel).to1();}
    ...
    default T_k getf_k(){return this.apply((λx_1 ... x_k.(λ.x_k)^CFromk)^CSel).tok();}
    CUnion apply(CSel s);
}
```

The translation of expressions is given below.

Set $xv ::= x \mid v$.
$[\![x]\!] = x$
$[\![e.m(e_1, \dots, e_n)]\!] = [\![e]\!].m([\![e_1]\!], \dots, [\![e_n]\!])$
$[\![(\lambda xs.e)^I]\!] = (\lambda xs.[\![e]\!])^I$
$[\![\texttt{new}\ C()]\!] = (\lambda x.x)^C$
$[\![\texttt{new}\ C(xv)]\!] = (\lambda.[\![xv]\!])^C$
$[\![\texttt{new}\ C(xv_1, \dots xv_n)]\!] = (\lambda \texttt{s.s.apply}([\![xv_1]\!], \dots, [\![xv_n]\!]))^C$
$[\![e.f]\!] = [\![e]\!].\texttt{get}f\texttt{()}$

The translation preserves typing and semantics, as formally stated below.

▶ **Theorem 3.** *If* $\vdash P$ *and* $\emptyset \vdash_P e : T$*, then the following hold:*

1. $\vdash [\![P]\!]$.
2. $\emptyset \vdash_{[\![P]\!]} [\![e]\!] : T$.
3. $e \Rightarrow_P v^\infty$ *iff* $[\![e]\!] \Rightarrow_{[\![P]\!]} [\![v^\infty]\!]$*, where* $[\![\infty]\!] = \infty$.

Proof. The first two points can be proved by straightforward induction on the typing rules. The third point is a consequence of the following properties:

- e is a value iff $[\![e]\!]$ is a value
- if $e \to_P e'$, then $[\![e]\!] \to^\star_{[\![P]\!]} [\![e']\!]$

The first property trivially holds, since e is closed, whereas the second one can be proved by structural induction on e. We show the most interesting case, which is $e.f$. There are the following subcases:

- Rule (CTX) is applicable, hence we have $e \to_P e'$ and $e.f \to_P e'.f$. By inductive hypothesis, $[\![e]\!] \to^\star_{[\![P]\!]} [\![e']\!]$ in n steps. Moreover, $[\![e.f]\!] = [\![e]\!]\texttt{.get}f\texttt{()}$, which is of shape $\mathcal{E}[[\![e]\!]]$, then the thesis follows by applying n times rule (CTX).
- Rule (FIELD) is applicable. We distinguish the following subcases:
  - $n = 1$: we have $\texttt{new}\ C(v).f \to_P v$ and $\mathsf{fields}_P(C) = T\ f\texttt{;}$; moreover, $[\![\texttt{new}\ C(v).f]\!] = (\lambda.)^C\texttt{.get}f\texttt{()}$. By the second clause in the translation of classes, class C exists in $[\![P]\!]$ and has a unique abstract method $\texttt{get}f$, hence $(\lambda.)^C\texttt{.get}f\texttt{()} \to_{[\![P]\!]} [\![v]\!]$ by rule (β).
  - $n > 1$: we have $\texttt{new}\ C(v_1 \dots v_n).f \to_P v_i$ and $\mathsf{fields}_P(C) = T_1\ f_1\texttt{;} \dots T_n\ f_n\texttt{;}$. Moreover, $[\![\texttt{new}\ C(v_1 \dots v_n).f]\!] = (\lambda\texttt{s.s.apply(}[\![v_1]\!], \dots, [\![v_n]\!]\texttt{)})^C\texttt{.get}f_i\texttt{()}$. By the third clause in the translation of classes, classes C, $C\texttt{From}T_i$, and $C\texttt{Sel}$ exist in $[\![P]\!]$ with the specified methods.
    Hence, $(\lambda\texttt{s.s.apply(}[\![v_1]\!], \dots, [\![v_n]\!]\texttt{)})^C\texttt{.get}f_i\texttt{()} \to^\star_{[\![P]\!]} [\![v_i]\!]$ by the following reduction sequence in $[\![P]\!]$, where we write the computational rule applied at each step:

    $(\lambda\texttt{s.s.apply(}[\![v_1]\!], \dots, [\![v_n]\!]\texttt{)})^C\texttt{.get}f_i\texttt{()} \to$ (INVK)

    $(\lambda\texttt{s.s.apply(}[\![v_1]\!], \dots, [\![v_n]\!]\texttt{)})^C\texttt{.apply((}\lambda x_1 \dots x_n.(\lambda.x_i)^{C\texttt{From}T_i}\texttt{)}^{C\texttt{Sel}}\texttt{).to}i\texttt{()} \to$ (β)

    $(\lambda x_1 \dots x_k.(\lambda.x_i)^{C\texttt{From}T_i})^{C\texttt{Sel}}\texttt{.apply(}[\![v_1]\!], \dots, [\![v_n]\!]\texttt{).to}i\texttt{()} \to$ (β)

    $(\lambda.[\![v_i]\!])^{C\texttt{From}T_i}\texttt{.to}i\texttt{()} \to$ (β)

    $[\![v_i]\!]$ ◀

4 Programming with only lambdas

We describe how this programming paradigm can be effectively used. In these examples, we assume a language extended with numbers and string literals, local variable declarations, type inference for lambdas and generics as in Java. First of all, to avoid to have to manually encode tuples by lambdas, we expect this encoding to be provided by libraries. That is, we expect to have library types `Tuple2<T1,T2>`, `Tuple3<T1,T2,T3>`, and so on. In this way, the programmer can just extend the opportune `Tuple` interface to encode a class with many fields.

```
//Library code:
//case for 2 fields
interface Union2<T1,T2>{
  default T1 to1(){ /*error*/}
  default T2 to2(){ /*error*/}
  }
interface From2_1<T1,T2> extends Union2<T1,T2>{ T1 to1(); }
interface From2_2<T1,T2> extends Union2<T1,T2>{ T2 to2(); }
interface Tuple2<T1,T2>{
  Union2<T1,T2> apply(Sel2<T1,T2> sel);
  default T1 get1(){
    return this.apply(
      (x1,x2)->(From2_1<T1,T2>)()->x1
```

```
      ).to1();
  }
  default T2 get2(){/*as above, using From2_2/to2()*/}
  }
interface Sel2<T1,T2>{ Union2<T1,T2> apply(T1 f1, T2 f2); }

...//case for 3 or more fields

interface Tuple0 { void dummy(); }

interface Function<T,R>{ R apply(T t);}//as in Java 8
interface Bifunction<T1,T2,R>{ R apply(T1 t1, T2 t2);}
//... other functional interfaces
```

With such simple standard library, we can write an extended version of the example in the introduction as follows:

```
interface Person extends Tuple2<String,Integer>{
  default String name(){return this.get1();}
  default Integer age(){return this.get2();}
  default String greet(){return "Hi, I'm "+this.name();}
  ...
  }
Bifunction<String,Integer,Person> makePerson
  = (name,age)->s->s.apply(name,age);
Person bob = makePerson.apply("bob",23);
bob.greet();
```

The code above shows that it is easy to encode classes with fields and instantiate them. It is also easy to abstract over object creation and provide factories, as the `Bifunction` above.

In the second example, we show how we can encode private methods; a library may want to encapsulate the richer `ConcretePoint` implementation and only expose a minimal `Point` interface.

```
//Point library code:
/**Point docs*/
interface Point{Integer x(); Integer y(); Integer distance(Point p);}
/**ConcretePoint is only for internal library usage
and may change in a future release*/
interface ConcretePoint extends Point, Tuple2<Integer,Integer>{
  //point implemented with a Tuple
  default Integer x(){return this.get1();}
  default Integer y(){return this.get2();}
  default Integer distance(Point p){/*uses aux*/}
  default Integer aux(Point p){..} //"private" method
  }
/**NewPoint docs do not need to mention ConcretePoint*/
interface NewPoint extends Tuple0, Bifunction<Integer,Integer,Point>{
  default Point apply(Integer x,Integer y){
    return (ConcretePoint) s->s.apply(x,y);
    }
  }
//User code
Point myPoint = ((NewPoint)()->{}).apply(3,5);
myPoint.distance(myPoint);//ok
//myPoint.aux(myPoint);//no
```

The interface `NewPoint` extends `Tuple0`, leaving the `dummy` method abstract, hence, as explained before, is expected to be instantiated by the dummy lambda. Moreover, it extends `BiFunction`, providing an implementation for the abstract method `apply`.

Note how implementation hiding is encoded by using subtyping and factories. This approach is a natural extension of what is discussed in TraitRecordJ [3], where methods of classes are implicitly private, and the only way to publicly expose behaviour is to implement an interface method.

Likely, the interface `ConcretePoint` should not be public. However, even with `ConcretePoint` exposed to the user, the implementation of `Point` is still hidden, and the user in no way can be aware that a `Point` is indeed a `ConcretePoint`, or call the `ConcretePoint.aux` method on an object provided by `NewPoint`.

Finally, we show that in this programming style recursive types are as easy as in plain Java. This is important to notice, since in other encodings of objects mutually recursive types are not trivial to handle. Consider the standard implementation of lists as pairs consisting of head and tail.

```
interface List<T>{
  T head();
  List<T> tail();
  boolean isEmpty();
  default  List<T> add(T elem){
    return (Cons<T>)s->s.apply(elem,this);
  }
}
interface Empty<T> extends List<T>,Tuple0 {
  default T head(){/*error*/}
  default List<T> tail(){/*error*/}
  default boolean isEmpty(){return true;}
}
interface Cons<T> extends Tuple2<T,List<T>>, List<T> {
  default T head(){return this.get1();}
  default List<T> tail(){return this.get2();}
  default boolean isEmpty(){return false;}
}
...//usage example
List<Integer> list = ((Empty<Integer>)()->{}).add(1).add(2).add(3);
```

Interestingly, we can also easily encode the iconic functional match:

```
interface List<T>{
  <R> R match(Supplier<R> onEmpty, Bifunction<T,List<T>,R> onCons);
  default  List<T> add(T elem){
    return (Cons<T>)s->s.apply(elem,this);
  }
}
interface Empty<T> extends List<T>,Tuple0 {
  <R> R match(Supplier<R> onEmpty, Bifunction<T,List<T>,R> onCons){
    return onEmpty.get();
  }
}
interface Cons<T> extends Tuple2<T,List<T>>, List<T> {
  <R> R match(Supplier<R> onEmpty, Bifunction<T,List<T>,R> onCons){
    return onCons.apply(this.get1(),this.get2());
  }
}
```

```
...//example methods using match
default Integer sumAll(List<Integer> l){
  return l.match(
    ()->0,
    (head,tail)->head+sumAll(tail)
  );
}
default Integer getHead(List<Integer> l, Integer orElse){
  return l.match(
    ()->orElse,
    (head,tail)->head
  );
}
```

The above example employs a variant of the visitor pattern [5] getting popular in Java 8, which provides functions as arguments to the visitor. The point to be noted here is that the λ-based paradigm makes this approach very natural. Note that `List` as defined above is open: any user can define new kinds of `List` by providing an implementation for the `match` method.

We end this section with a remark. A subtle detail in Java 8 is that lambdas cannot be used to implement generic methods. This means that the following more natural tuple encoding would only work by "desugaring" the lambda syntax.

```
public interface Tuple2<A,B> {
  <T> T apply(BiFunction<A, B, T> f);
  default A a(){return apply((a,b)->a);}
  default B b(){return apply((a,b)->b);}
}
...//code trying to instantiate Tuple2
default <A, B> Tuple2<A,B> of(A a, B b){
    //return f -> f.apply(a, b);//does not compile :-(
    return new Tuple2<A,B>(){//works without lambda syntax
      public <T> T apply(BiFunction<A,B,T> f){
        return f.apply(a,b);}};}
```

Allowing lambdas to implement generic methods was planned in a pre-release of Java 8, but this feature was removed before release. Note how the generic method `match` above is the only abstract method of `List`, thus if generic methods were instantiable with lambdas, we could omit the `Cons` code and simply write

```
interface List<T>{
  <R> R match(Supplier<R> onEmpty, Bifunction<T,List<T>,R> onCons);
  default List<T> add(T elem){
    return (onEmpty,onCons)->onCons.apply(elem,this);
  }
}
```

5 Discussion and related work

The encoding presented in Section 3 strikes a balance between being not too cumbersome and useful. A nice feature is that it does not change the way that the programmers write their program. That is, it is not a transformation that turns the program inside-out and obscures the original intent: the original classes are still there (as interfaces) and their construction

happens at the same sites (with lambda syntax). On the other hand, the representation of object's state as a tuple is mainly intended to show the completeness of the approach, and could be replaced by an efficient private implementation as, e.g., values of primitive types are seen as objects in Smalltalk. The translation is shown over a simplification of the FJ calculus, which makes it composable with other work on Java semantics. Keeping the original program structure (classes/methods) means that the original program is still extensible in the same way. Also, Java reflection would still work with the output of this encoding in a natural way. So, this encoding looks quite powerful to support more features, e.g., simulate field shadowing or more complex inheritance patterns.

Since our motivation was to explore a programming style which is basically *functional*, we did not consider imperative features, as it is in FJ. They can be added, as usually in functional languages, introducing reference types and constructs for referencing, assigning and dereferencing. The classical encoding of tuples by functions immediately extends, as shown by the code below, which is the previous Haskell example rewritten in OCaml with references:

```
type a = int
type b = bool
type aOrBref = FromA of a ref | FromB of b ref
type fst = a ref -> b ref -> a ref
type snd = a ref -> b ref -> b ref
type sel = FromFst of fst | FromSnd of snd
type pair = sel -> aOrBref

let mypair : pair =
  let first = ref 1 in
  let second = ref true in
    function
      FromFst s -> FromA (s first second)
      | FromSnd s -> FromB (s first second)

let getFst : pair -> a =
  function p ->
    let FromA a = p (FromFst (function a -> function b -> a))
    in !a
let setFst : pair -> a -> unit =
  function p -> function x ->
    let FromA a = p (FromFst (function a -> function b -> a))
    in a; a := x;
```

In a Java-like language, the same can be achieved having reference types (as in C++), or, otherwise, by using local variables. Unfortunately, in Java local variables used in a lambda expression must be final or effectively final. Without this restriction, a class Person with an updatable field `name` could be encoded as follows:

```
interface Person extends Tuple2<Supplier<String>, Consumer<String>>{
  default String getName(){return this.get1().apply();}
  default void setName(String name){return this.get2().apply(name);}
}
interface PersonFactory{
  void dummy();
  default Person makePerson(String name){
    return  s -> s.apply(()->name, newName->name=newName);
  }
}
```

One commonly used way to circumvent this Java limitation is to use an array of size 1, as shown below:

```
default Person makePerson(String name){
  String[] n={name};
  return  s -> s.apply(  ()->n[0],    newName->n[0]=newName  );
}
```

In general, adding any kind of array or collections would make our encoding simpler, but, as said above, here our goal is to achieve minimality.

Though, as already said, addressing limitations of existing programming techniques was not our aim, we can mention some side benefits of the approach. Replacing "real" fields by getters/setters avoids the limitation that the type of fields must be invariant; thus more code reuse patterns become available, see the extended discussion in [12]. Moreover, Java8 interfaces support reusing code from multiple sources, as for multiple inheritance. Avoiding classes means that all the code is usable for multiple inheritance.

We already mentioned that inheritance and dynamic dispatch are supported through default methods in interfaces. In addition, we could easily extend our core to support `InterfaceName.super.methName(...)` and this would transparently allow super method calls as in Java. Moreover, the above syntax could be syntactic sugar; if any interface declared methods with a standard long name `returnType InterfaceName$methName(...){...}` with a delegator method `returnType methName(...){ return this.InterfaceName$methName(...);}` then `InterfaceName.super` could be emulated simply using the longer name for the method.

An OO style without class declarations, called interface-based object-oriented programming, has been proposed in [12], and exploited in Java by defining *Classless Java*. However, differently from our proposal, Classless Java has objects, obtained as instances of anonymous inner classes, hence fields as well, and `static` methods.

More generally, many languages support objects without classes, that is, follow the so-called "object-based" paradigm. That is, objects are not created as class instances, but, e.g., by directly writing "object literals". Our proposal goes a bit further, since in our calculus there are no objects at all; the *only values* are functions (lambdas).

Differently from the Java approach, in [11] a minimal core Java is extended to λ-expressions by adding function types, following the style of functional languages. Complexity of type inference increases in a substantial way, with respect to Java's one. Moreover, adding real function types entails that a method must have a different signature according to whether it can accept an object or a function. This contrasts with Java philosophy to fuse language innovations into the old layer.

Empirical methodologies are used in [10] to illustrate when, how and why imperative programmers adopt λ-expressions.

Classical encodings of objects [4] are as records of mutually recursive functions, where all such functions are closures over the record. In this kind of encoding, objects contain all their behaviour, and a program is fully expressed by an expression. Typing in this setting is non trivial, especially recursive types, which need to be handled by some variation of fixpoints. Our proposed language, instead, embraces the idea of an externally defined table of types (in our case interface names) also including most of the behaviour, in the form of default implementations. In this way, the key technical characteristic of OOP following, e.g., [1], that is, dynamic dispatch, is provided anyway, and such language design is more friendly toward module systems and module composition languages.

Grace [9, 7] offers an interesting middle ground: it is structurally typed, but objects can have (generic) type-alias declarations as members, and these type names can be mutually recursive. In most Grace programs, a top level object (called a *module*) plays the role of

the table of types, and reduction actually takes place inside such a module object. Still, the Grace approach is more flexible than having a fixed top-level type table, since multiple objects can be nested into each other, and lexical scope and nesting allow for interesting forms of code composition. However, to make static reasoning feasible, only type members of *known objects* (objects created in a controlled way) can be used as type annotations. Moreover, due to generic type aliases, subtyping is undecidable.

Formalization of lambdas as in Java 8 have been provided in [2, 6], the former covering intersection types and default methods as well. These works focus on typing issues, notably on the fact that lambdas can only be typed when occurring in a context requiring a given type (called the *target type*). In a small-step semantics, this poses a problem: reduction can move λ-abstractions into arbitrary contexts, leading to intermediate terms which would be ill-typed. To maintain subject reduction, in [2] λ-abstractions are decorated with their initial target type. In a big-step semantics, as in [6], there is no need of such intermediate terms and annotations.

6 Conclusion

We have described a novel way to conciliate OO and functional programming, where objects (instances of classes) are replaced by tuples (encoded by lambdas). Lambdas can be equipped with an additional behaviour, thanks to the fact that they may implement interfaces with default methods, hence inheritance and dynamic binding are still supported. The encoding has been formally defined by a translation from a calculus including classes to one with only lambdas and interfaces, shown to preserve typing and semantics. This novel programming style has been illustrated by several examples.

Concerning further work, a first step is about the use of the λ-based paradigm in Java 8, illustrated in Section 4. We assumed the encoding of tuples by lambdas to be provided once and for all by libraries. However, in the examples in Section 4, the programmer still has to manually write some tedious and rather cryptic code, notably lambdas such as `(name,age)->s->s.apply(name,age)`, or dummy lambdas. To make the paradigm more user-friendly, suitable syntactic sugar should be provided for these constructions, likely in form of macros.

More in general, we could leave the Java world and investigate the design of a language especially suited to express this paradigm, and its integration with other typical language features. For instance, which would be the best, simplest way to encode mutation in $\mathrm{FJ}^{-}_{\lambda}$ where fields are implicit? An inspiration could come from Smalltalk, which allows to update local variables, even when they are captured by a closure.

In this design investigation, in particular an interesting direction is to take an approach which is complementary to that of this paper, that is, to start from a functional kernel and to enrich it by a table of types.

References

1 Jonathan Aldrich. The power of interoperability: why objects are inevitable. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH'13*, pages 101–116. ACM Press, 2013. doi:10.1145/2509578.2514738.

2 Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Java & Lambda: a Featherweight story. *Logical Methods in Computer Science*, 14(3), 2018. doi:10.23638/LMCS-14(3:17)2018.

3 Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocco. TraitRecordJ: A programming language with traits and records. *Science of Computer Programming*, 78(5):521–541, 2013. doi:10.1016/j.scico.2011.06.007.

4 Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1-2):108–133, 1999. doi:10.1006/inco.1999.2829.

5 Peter Buchlovsky and Hayo Thielecke. A type-theoretic reconstruction of the visitor pattern. *Electronic Notes in Theoretical Computer Science*, 155:309–329, 2006. Mathematical Foundations of Programming Semantics - MFPS 2005. doi:10.1016/j.entcs.2005.11.061.

6 Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. Soundness conditions for big-step semantics. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming - ESOP 2020*, volume 12075 of *Lecture Notes in Computer Science*, pages 169–196. Springer, 2020. doi:10.1007/978-3-030-44914-8_7.

7 Michael Homer, Timothy Jones, and James Noble. First-class dynamic types. In *Dynamic Languages Symposium 2019*, pages 1–19. ACM Press, 2019. doi:10.1145/3359619.3359740.

8 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

9 Timothy Jones, Michael Homer, James Noble, and Kim Bruce. Object inheritance without classes. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *ECOOP'16 - Object-Oriented Programming*, volume 56, pages 13:1–13:26. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.13.

10 Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the use of lambda expressions in Java. *Proceedings of ACM on Programming Languages*, 1(OOPSLA 2017):85:1–85:31, 2017. doi:10.1145/3133909.

11 Martin Plümicke. Well-typings for Java$_\lambda$. In Christian W. Probst and Christian Wimmer, editors, *Principles and Practice of Programming in Java - PPPJ 2011*, pages 91–100. ACM Press, 2011. doi:10.1145/2093157.2093171.

12 Yanlin Wang, Haoyuan Zhang, Bruno C. d. S. Oliveira, and Marco Servetto. Classless Java. In Bernd Fischer and Ina Schaefer, editors, *Generative Programming: Concepts and Experiences - GPCE 2016*, pages 14–24. ACM Press, 2016. doi:10.1145/2993236.2993238.

Multiparty Languages: The Choreographic and Multitier Cases

Saverio Giallorenzo
Università di Bologna, Italy
INRIA, Sophia Antipolis, France

Fabrizio Montesi
University of Southern Denmark, Odense, Denmark

Marco Peressotti
University of Southern Denmark, Odense, Denmark

David Richter
Technical University of Darmstadt, Germany

Guido Salvaneschi
University of St. Gallen, Switzerland

Pascal Weisenburger
University of St. Gallen, Switzerland

Abstract

Choreographic languages aim to express multiparty communication protocols, by providing primitives that make interaction manifest. Multitier languages enable programming computation that spans across several tiers of a distributed system, by supporting primitives that allow computation to change the location of execution. Rooted into different theoretical underpinnings – respectively process calculi and lambda calculus – the two paradigms have been investigated independently by different research communities with little or no contact. As a result, the link between the two paradigms has remained hidden for long.

In this paper, we show that choreographic languages and multitier languages are surprisingly similar. We substantiate our claim by isolating the core abstractions that differentiate the two approaches and by providing algorithms that translate one into the other in a straightforward way. We believe that this work paves the way for joint research and cross-fertilisation among the two communities.

2012 ACM Subject Classification Software and its engineering → Multiparadigm languages; Software and its engineering → Concurrent programming languages; Software and its engineering → Distributed programming languages

Keywords and phrases Distributed Programming, Choreographies, Multitier Languages

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.22

Category Pearl

Funding *Fabrizio Montesi*: Villum Fonden, grant no. 29518, and Independent Research Fund Denmark, grant no. 0135-00219.
David Richter: German Federal Ministry of Education and Research (BMBF) iBlockchain Project, grant no. 16KIS0902.
Guido Salvaneschi: German Research Foundation (DFG) project no. 383964710, LOEWE initiative (Hesse, Germany) within the Software-Factory 4.0 project, and Swiss National Science Foundation (SNSF) project "Multitier Programming above the Clouds".
Pascal Weisenburger: University of St. Gallen, IPF project no. 1031569.

Acknowledgements We thank the anonymous reviewers for their useful feedback and comments.

1 Introduction

Programming concurrent and distributed systems is notoriously hard. Among other issues, it requires dealing with coordination and predicting how multiple participants will interact at runtime, for which programmers do not receive adequate help from mainstream programming abstractions and technology [25, 21, 32].

The quest for finding elegant languages and methodologies that can help with concurrent and distributed programming has been a major focus of the research community for decades, including the seminal actor model and calculus of communicating systems [17, 27]. In this work, we are interested in two kinds of languages that have been recently gaining attention: *choreographic languages* [28, 2] and *multitier languages* [40]. Choreographic languages are designed to express multiparty communication protocols, by providing primitives that make interaction manifest. On the other hand, multitier languages allow for programming computation that spans across several tiers of a distributed system, by providing primitives that allow computation to change location of execution.

Both choreographic and multitier languages aim at making concurrent and distributed programming more effective, and have inspired several research and industrial language designs. However, choreographic and multitier languages stem from different ideas; they adopt different terminologies; they look different; they have evolved different features; and they have found different applications in practice. Perhaps because the design principles of choreographic and multitier languages come from different angles, the two communities have prolifically evolved independently. However, as a consequence, the commonalities and actual differences between the two research lines remain unclear, which impedes cross-fertilisation.

In this paper, we offer a new perspective on the relationship between choreographic and multitier languages. We show that, despite their different starting points and evolutions, they share a strong core idea that classifies them both as what we call *multiparty languages* – languages that describe the behaviour of multiple participants. Leveraging this commonality, it is possible to derive choreographic programs from multitier programs, and vice versa. Our aim is to provide a way for each community to access the other, encouraging cross-fertilisation.

We outline our investigation and contributions:

- In Section 2, we give an overview of the essential features of choreographic and multitier languages. We recap the history of the two approaches and identify their key differences, which lie in perspective (objective vs subjective) and in the modelling of communications (manifest vs non-manifest). We also pinpoint the commonality that classifies choreographic and multitier languages as multiparty.
- In Section 3, we present an example use case for both choreographic and multitier programming, which introduces the concrete choreographic and multitier programming languages that we will use in the rest of our development: Choral [16] and ScalaLoci [38].
- In Section 4, we introduce Mini Choral and Mini ScalaLoci, two representative but minimal languages for choreographic and multitier programming, respectively. Mini Choral and Mini ScalaLoci dispense with the features that are not essential parts of their respective paradigms, which allows us to study how the essential differences can be bridged in the next section.
- In Section 5, we define algorithms for translating programs in Mini Choral to programs in Mini ScalaLoci, and vice versa. The translations deal with the changes in perspective and manifestation of communications between the two paradigms. For example, translating a multitier program into a choreographic one requires synthesising a communication protocol that enacts the necessary communications among participants.

Our translations are not just of inspiration to see the connection between the two paradigms (which we leverage in the next section), but also open a window towards the future sharing of theoretical and practical results. An example for each direction: by translating a multitier program into a choreographic one and then using a choreographic compiler to generate executable code, we can know statically the pattern of communications that will be enacted by the executable code (this property is called "Choreography Compliance" [16] or "EndPoint Projection Theorem" [3]); by translating a choreographic program into a multitier one and then using a multitier compiler to generated executable code, we can reuse all the machinery developed by the multitier community to generate code for different technologies (e.g., the code generated for one participant is in JavaScript for a web browser while the code for another might be code runnable on the Java Virtual Machine for a server).

- Our study shows that, while choreographic and multitier programming languages are different enough to be independently useful, they are also near enough to benefit from cross-fertilisation. In Section 6, we report on important features that have been developed separately in the choreographic and multitier research lines. We find that important features for the development of concurrent and distributed systems have been developed for one paradigm but not the other. Inspired by our newfound connection, we discuss how these features could be ported over to the other paradigm in the future, setting up future work enabled by our view.

2 Background: Choreographic and Multitier Programming Languages

In this section, we give some background on choreographic and multitier languages, and discuss their differences and similarities.

2.1 Choreographic Languages

Choreographic languages are inspired by the famous "Alice and Bob" notation, or security protocol notation [30]. The idea is to define how the different participants of a system should communicate (or interact) – which later inspired also message sequence charts and sequence diagrams [20]. Textual and graphical choreographic languages have already been adopted in industry as specification languages in different settings ranging from business processes, e.g., the choreographic language in OMG's Business Process Model and Notation, to web services, e.g., W3C's Web Services Choreography Description Language [31, 37].

The essence of a choreographic language is the capability of expressing explicitly data flows from a participant to another through communication, and of composing such communications into larger structures. In other words, choreographies make interaction and the structure of interaction protocols *manifest*. A communication from a participant, `Alice`, to another, `Bob`, is written as follows:

```
Alice.userId -> Bob.x : ch
```

The statement above reads: `Alice` sends its `userId` (a local variable storing a user identifier) to `Bob`, which stores it in its local variable `x`, and the communication takes place through the channel `ch`.

Communication statements can be composed in larger and more sophisticated protocols, for example using the sequential operator "`;`". In the following protocol snippet: after interacting with `Alice`, `Bob` forwards to `Charlie` the user identifier that it received through a separate channel `ch2`.

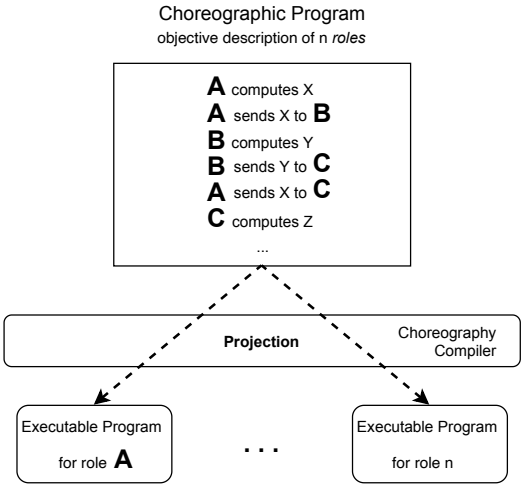


Figure 1 Choreographic Programming.

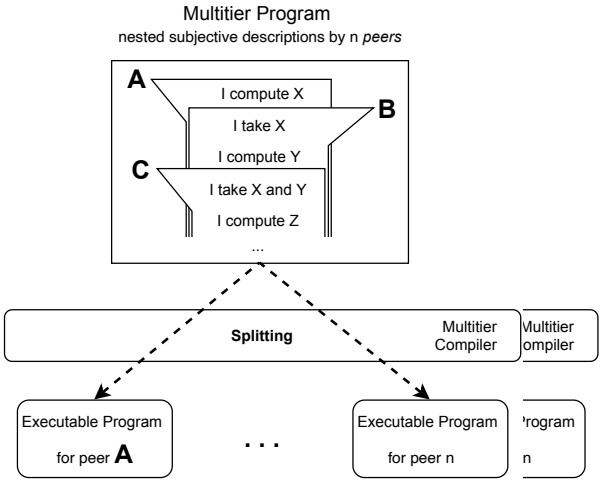


Figure 2 Multitier Programming.

Listing 1 A simple choreography with three participants.

```
1 Alice.userId -> Bob.x : ch;
2 Bob.x -> Charlie.y : ch2;
```

In the paradigm of *choreographic programming* [28], choreographic languages are full-fledged programming languages: developers write the implementation of an entire multiparty system as a choreography, and then a compiler automatically generates an executable program for each participant. This process is depicted in Figure 1. Choreographies resemble play scripts, written from an external point of view, describing the interactions among all participants. We call this view *objective*. Participants, like `Alice` and `Bob`, are typically referred to as *roles* in choreographies, and the procedure that generates the executable program for each role is called projection (or endpoint projection) [4, 11].

The code in Listing 1 is valid code in the Chor language, the first implementation of choreographic programming [28, 4]. Chor targets microservices: given that code (with appropriate boilerplate), Chor would generate executable programs of microservices that implement `Alice`, `Bob`, and `Charlie`. Choreographic programming has been applied to other settings, e.g., information flow [22], parallel algorithms [10], cyber-physical systems [24, 23], runtime adaptation [11], and integration processes [15].

2.2 Multitier Languages

Multitier languages are inspired by one of the ideas proposed with ambient calculi [5]. In this kind of process calculi, terms express the place (the "ambient") at which computation occurs. Computations that take place at different locations can be nested, which enables describing multiparty systems. It was later shown that the idea can be combined with well-known abstractions, by developing a variation of λ-calculus with locations called Lambda 5 [29]. This solution prompted the development of *multitier languages* [36, 8, 40], which extend existing programming languages with locations. The term multitier comes from the fact that these languages were mostly developed for web programming, where tiers is used to refer to the typical participants of a web system (e.g., client, backend server, and database).

The crux of a multitier language is the capability of hopping from the point of view of a participant to that of another – the multitier language by Serrano et al. is aptly called "Hop" [36]. When hopping from a participant to another, it is possible to move data from the participant that we are leaving to the participant that we are going to – enabling communication. As an example, consider a remote procedure call from a client to server. In a recent incarnation of multitier programming that builds on the Scala language, ScalaLoci [38], this can be written as follows (for simplicity of presentation, we omit library calls that would be necessary to deal with asynchrony):

```
def rpc(input: String): String on Client = on[Client] {
  val result =
    on[Server].run.capture(input) {
      expensiveFunction(input)
    }.asLocal
  return result
}
```

Participants are referred to as peer types in ScalaLoci. The method `rpc` above is defined as a block of code that starts at the client peer (`on[Client]`). The client stores the result of some computation in its local variable `result`, but this computation is performed at the server. This result is achieved by "moving" to the server with the instruction `on[Server]`. The invocation of method `run`, right afterwards, models some computation, and `capture(input)` means that we want to move the content of the local variable `input` from the peer that we are leaving (the client) to the one that we are going to (the server). How this move is achieved is left to the implementation (ScalaLoci generates a communication strategy automatically). The server then runs an expensive function on the input, and the execution goes back to the client – the code block at the server ends. The invocation of `asLocal` ensures that the return value of the code at the server is moved to the location of the enclosing scope (the client). We finally return the result at the client.

Like choreographic programming languages, multitier languages come with a compiler that turns the multiparty view of the system into executable programs. This process is depicted in Figure 2. Given a multitier program, a multitier compiler generates an executable program for each peer type (in the case of Section 2.2, these would be client and server). The procedure for generating code is called splitting. The nested "dialogues" of peers inside the multitier program depict that a multitier program has many viewpoints, switching regularly from the point of view of a peer to that of another. Nevertheless, code is written with the viewpoint of the peer we are currently in. For this reason, we say that multitier programs adopt a nested *subjective* view.

2.3 Towards Linking Choreographic to Multitier Languages

The two communities of choreographic and multitier languages have prolifically evolved independently [2, 40]. They adopted different design principles, and they have found different practical applications – most notably service-oriented computing for choreographies and web development for multitier programming. As a result, they have also developed several features independently (we discuss some of the most important ones in Section 6). In addition, the two communities have been facing different challenges. For example, multitier programming languages historically tackle the problem of "impedance mismatch": the necessity of handling data conversions and heterogeneous execution engines in the web (the Google Web Toolkit is a multitier framework that contributes to this research area). Instead, choreographic programming mainly aimed at achieving "choreography compliance": providing the guarantee that distributed systems communicate as expected and with desirable properties (like liveness).

Yet, the two paradigms are clearly linked. We drew Figure 1 and Figure 2 with the intention of highlighting such connection. Indeed, despite differences in both terminologies and methods, the strategies of choreographic and multitier programming languages share a similarity: both define the behaviour of a multiparty system in a single compilation unit, and then offer ways to synthesise executable implementations for the participants. We thus identify both kinds of languages as instances of the larger class of *multiparty languages* – leaving the class open to future additions. We see value in both techniques for multiparty programming. In choreographies protocols are manifest, which makes them easy to understand. Multitier programs give access to multiparty programming with a developing experience that resembles standard "local programming" by leveraging scoping.

Despite both choreographic and multitier languages sharing the multiparty approach, they remain pretty diverse in terms of theoretical background. The theory of choreographic language typically stands on process calculi, whereas multitier models build on λ-calculus [18, 4, 19, 11, 40]. This is likely an important reason why the link between choreographic and multitier languages has been overlooked for long. Very recently, however, it has been shown that object-oriented languages can be extended to capture choreographies, by generalising the notion of data type to data types located at *multiple roles* [16]. In the resulting language, called Choral, a choreography among a few roles can be expressed as an object. For example, we can write the choreography in Listing 1 in Choral as follows:

```
1  class Example@(Alice, Bob, Charlie) {                // the three roles of the protocol
2    DiDataChannel@(Alice,Bob)<Serializable> ch;        // channel from Alice to Bob
3    DiDataChannel@(Bob,Charlie)<Serializable> ch2;     // channel from Bob to Charlie
4
5    /* constructor omitted */
6
7    public UserID@Charlie run(UserID@Alice userId) {   // the protocol
8      UserID@Bob x = ch.<UserID>com(userId);           // Alice.userId -> Bob.x : ch
9      return ch2.<UserID>com(x);                       // Bob.x -> Charlie.y : ch2
10   }
11 }
```

Briefly – as we give a more detailed description of Choral programs in Section 3.2 – the `Example` class declares three roles (**`Alice`**, **`Bob`**, and **`Charlie`**) and two directed channels (`ch` from **`Alice`** to **`Bob`** and `ch2` from **`Bob`** to **`Charlie`**). These correspond to the roles and channels assumed in Listing 1. The protocol described in Listing 1 is implemented by method `run` that takes an instance of `UserID` located at **`Alice`** and returns one located at **`Charlie`** passing through **`Bob`**. Communication happens by invoking method `com` of the two channels.

Choral helps in leveling the playfield with multitier programming. Indeed, we now have an object-oriented incarnation of choreographic programming that we can use to compare to object-oriented multitier languages, here represented by ScalaLoci. In the next sections, we leverage this common ground and take Choral and ScalaLoci as representative languages for their respective paradigms.

3 Overview of Choral and ScalaLoci

In this section, we give an overview of the representative languages for choreographic and multitier programming that we have chosen, Choral and ScalaLoci, by using them to deal with a simple yet comprehensive example of a context-aware protocol for e-mail fetching.

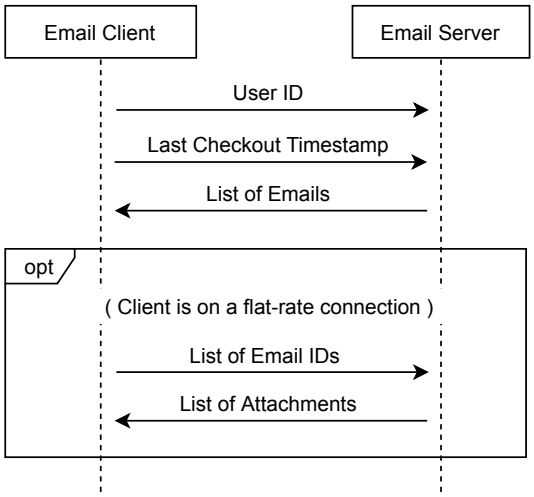


Figure 3 Sequence diagram for context-aware e-mail fetching.

3.1 A Context-Aware Email-Fetching Protocol

Before delving into the details of the two implementations, we discuss briefly the protocol that we want to program. A depiction as a sequence diagram is given in Figure 3. The protocol defines an interaction between an Email Client and an Email Server. Specifically, the Client sends its identification token – here simplified as User ID – and the timestamp of the last e-mail checkout to the Server. The Server returns the list of e-mails received after the timestamp to the Client. After the above interaction, the Client and the Server enter an optional block. The optional block is executed depending on the context of the client, namely, if the connection from the Client to the Server is flat-rate, i.e., if the connection fee paid by the Client is independent from its usage. If that is the case, the Client sends the Server the list of e-mail IDs retrieved in the previous interaction to fetch their attachments. The Server concludes the optional part of the protocol by sending to the Client the requested attachments.

3.2 A Choreographic Programming Implementation with Choral

In Listing 2, we use Choral to implement the protocol from Figure 3. The example illustrates the main concepts of the choreographic programming approach and how Choral captures them in the object-oriented setting.

In Choral, objects have types of the form `T@(R1, ..., Rn)`, where `T` is the interface of the object (as usual), and `R1`, ..., `Rn` are the roles that collaboratively implement the object. As we see below, Choral supports two notations for denoting the roles over which an object is distributed: the standard form `@(A, ..., Z)` and the contracted form `@A`, for objects that belong to one role (shortcut for `@(A)`). Incorporating roles in data types makes distribution manifest at the type level.

In Listing 2, at Line 3, we define a class `EmailSystem` implemented by two roles: the `Client` and the `Server`. The method `updateEmails` (Line 8) implements the actual protocol from Figure 3. Lines 4–6 declare class-level `private` objects, i.e., accessible from the `updateEmails` method and other (omitted) ones within the class. Specifically, at Line 4, we have the `MailServerDB` located at the `Server`. At Line 5, we find the complementary `MailDB` of the `Client`. At Line 6, we define the object used to transfer data between the two roles: a `SymChannel` – standing for *symmetric channel* – shared between the two roles and able to transmit `Serializable` objects. We omit the initialisation of the abovementioned objects.

Listing 2 Choral implementation for the context-aware e-mail fetching example.

```
1  enum Choice@Role { THEN, ELSE }
2
3  class EmailSystem@(Client, Server) {
4    private MailServerDB@Server serverDB = ...;
5    private MailDB@Client clientDB = ...;
6    private SymChannel@(Client, Server)<Serializable> ch = ...;
7
8    void updateEmails(UserId@Client userId) {
9      UserId@Server id = userId >> ch::com;
10     Timestamp@Server timestamp = clientDB.lastCheckOut() >> ch::com;
11     List@Client<Email> emails = serverDB.since(id, timestamp) >> ch::com;
12     clientDB.update(emails);
13     if (ClientLib@Client.isOnFlatRate()) {
14       Choice@Client.THEN >> ch::select;
15       clientDB.extractIds(emails) >> ch::com
16         >> serverDB::getAttachments >> ch::com
17         >> clientDB::updateAttachments;
18     }
19     else {
20       Choice@Client.ELSE >> ch::select;
21     }
22   }
23 }
```

Considering the description of the implementation of the e-mail fetching protocol, we look at the `updateEmails` method (Line 8). The method does not return a value (`void`) and takes as input the `UserId` – which simplifies the user authentication procedure here, for brevity – to identify the user of the `Client` at the `Server`.

In the body, at Line 9, we pass the `UserId` to the `Server`. We do this by invoking the method `com` of the `ch SymChannel` giving to it as argument the `userId`. This is done by the expression `userId >> ch::com` which uses the Choral chaining operator `>>` and that corresponds to the expanded expression `ch.com(userId)`. To make Choral programs closer to standard choreographic notation, where data flows from left to right, Choral borrows the forward chaining operator `>>` from F#: `exp >> obj::method` is syntactic sugar for `obj.method(exp)`.

The method `com` of the `SymChannel` transfers the value of the sender given as input into an equivalent representation of the value at the receiver. In this case, the sender is the `Client` (where the `UserId` object lives) and the receiver is the `Server`, which stores the result of the communication into variable `id` which is an object of type `UserId` at its location – i.e., `UserId@Server`.

The transfer of the `Timestamp` from the `Client` to the `Server` is similar (Line 10): we retrieve the object from the `clientDB` – invoking method `lastCheckOut` – and we transfer it to the `Server` thought the `SymChannel`. Then, to fetch the e-mails, the `Client` receives a transmission from the `Server`. The `Server` interrogates its local database (`serverDB`) by extracting all e-mails belonging to the `id` of the `Client` and received `since` its last checkout (indicated by the `timestamp`) and sends them to the `Client` via their shared `SymChannel`. At Line 12, the `Client` uses the received list of `emails` to `update` its local database (`clientDB`).

Lines 13–20 implement the optional part of the protocol from Figure 3. First, the `Client` checks whether it is using a flat-rate connection – this is done through the static library `ClientLib` and its method `isOnFlatRate`.

The `if-else` block at Lines 13–20 allows us to explain the concept of knowledge of choice (a hallmark element of choreographic programming) and how Choral implements it. Briefly, the concept of knowledge of choice indicates a fork in the flow of a program among alternative behaviours, where the concerned roles should coordinate to ensure that they agree on which

behaviour they should enact. In choreographic languages, this issue is typically addressed by defining a "selection" primitive to communicate constants drawn from a dedicated set of "labels", so that the compiler has enough information to build code that can react to choices made by other roles [4, 11]. In Choral, this is implemented by channel methods that can transmit instances of enumerated types between roles. Conveniently, the `SymChannel` used in the example also supports selections via its `select` methods. In Listing 2, we find the implementation of the knowledge of choice of the conditional at Line 14 (where the `Client` "decides" to fetch the attachments) and at Line 20 (which skips the retrieval). In the example, we implement the choice by defining the `Choice enum` class at Line 1 – note that we use the identifier `Role` for the single role that owns the `Choice` object in its declaration, instantiated at the `Client` at Lines 14 and 20.

If the `Client` uses a flat-rate connection, the chained statement at Lines 15–17 execute: first (Line 15) the `Client` sends to the `Server` the IDs of the e-mails (retrieved through `extractIds(emails)`) whose attachments it wants to retrieve, then (Line 16) the `Server` uses the received `ids` to extract from its database (`serverDB`) the attachments and it send them back to the `Client`, and finally (Line 17) the `Client` uses the received attachments to update its local database.

3.3 A Multitier Programming Implementation with ScalaLoci

We now use ScalaLoci to illustrate the multitier programming approach, implementing the protocol from Figure 3 in Listing 3.

In ScalaLoci, the location of different values is specified through *placement types*. The placement type `T on P` represents a value of type `T` on a peer `P`. Developers can freely define the different components, called *peers*, of the distributed system. For instance, in the example, `serverDB` is a `MailServerDB` placed on the `Server` (Line 5) and `clientDB` is a `MailDB` placed on the `Client` (Line 6).

Peers are defined as abstract type members (Lines 2 and 3). Further, peer types express the architectural relation between the different peers by specifying ties between peers, thus supporting generic distributed architectures. Ties statically approximate the runtime connections between peers. In the example, we define a *single* tie from client to server (Line 2) and from server to client (Line 3). A single tie expresses the expectation that a single remote instance is always accessible. In the specified architecture, a client connects to a single server and a server program instance handles a single client.

The `updateEmails` method (Line 8) encapsulates the communication logic from Figure 3. It takes the `UserId` for identifying the client as input. The implementation diverts control flow to the server using a nested `on[Server].run` expression (Line 10). The `capture` clause transfers both the `timestamp` and the `userId` from the client to the server. Inside the server expression (Line 11), the server queries its local `serverDB` database to extract all e-mails belonging to the `userId` of the client received `since` its last checkout (indicated by the `timestamp`). The result of the server-side expression is returned to the client using `asLocal` (Line 12).

In ScalaLoci, accessing remote values via the `asLocal` marker creates a local representation of the remote value by transmitting it over the network. For simplicity, we use synchronous communication. In general, ScalaLoci allows developers to choose among different *transmitters*, most notably one that wraps local representations of data in futures to account for network delay and communication failures.

The client then uses the received list of `emails` to `update` its local `clientDB` database (Line 14). Lines 16–20 implement the optional part of the communication logic from Figure 3. If the client is currently using a flat-rate connection – as indicated by the static `ClientLib.isOnFlatRate`

Listing 3 ScalaLoci implementation for the context-aware e-mail fetching example.

```
1  @multitier object EmailSystem {
2    @peer type Client <: { type Tie <: Single[Server] }
3    @peer type Server <: { type Tie <: Single[Client] }
4
5    private val serverDB: MailServerDB on Server = ...
6    private val clientDB: MailDB on Client = ...
7
8    def updateEmails(userId: UserId): Unit on Client = on[Client] {
9      val timestamp: Timestamp = clientDB.latestCheckout
10     val emails: List[Email] = on[Server].run.capture(userId, timestamp) {
11       serverDB.since(userId, timestamp)
12     }.asLocal
13
14     clientDB.update(emails)
15
16     if (ClientLib.isOnFlatRate) {
17       val ids = clientDB.extractIds(emails)
18       clientDB.updateAttachments(
19         on[Server].run.capture(ids) { serverDB.getAttachments(ids) }.asLocal)
20     }
21   }
22 }
```

method – the client initiates a second server-side computations using `on[Server].run` (Line 19). The client transfers the IDs of the e-mails (retrieved through `extractIds(emails)`) – whose attachments to receive – to the server, which extracts the attachments from its `serverDB` database and returns them to the client, which then updates its local `clientDB` with the received attachments (Line 18).

4 Mini Choreographic and Multitier Languages

We now introduce Mini Choral and Mini ScalaLoci, minimal languages that omit most features of their reference counterparts that are irrelevant to our study (like generics and inheritance). This allows us to focus on the distinctive traits that characterise the choreographic and multitier approaches, respectively. The minimality of the two languages is instrumental to highlight their distinguishing features here and to focus on the salient points that define their reciprocal translations in Section 5. Next, we present the grammar of the two languages and briefly describe the components that mark them respectively as choreographic and multitier languages.

Listing 4 displays the grammar of Mini Choral. C ranges over class declarations, *Channel* ranges over channel declarations, *Field* ranges over class fields, *Method* ranges over method definitions, *Type* ranges over type expressions, and *Exp* ranges over expression terms. The metavariable *id* ranges over both class names, fields, and variables. We use `A`, `B`, `C` to range over role names. Here and in the reminder of the paper, we use $\overline{overlines}$ to denote sequences of terms of the same sort and we denote concatenation of sequences using a comma.

4.1 Mini Choral

The class declaration C defines its name *id*, its owner roles $\overline{\texttt{A}}$ within the `@`$(\cdots)$ clause, the topology of directed channels available between roles in $\overline{Channel}$, its field declarations $\overline{Field}$, and its suite of method definitions $\overline{Method}$.

Listing 4 Syntax of Mini Choral.

Mini Choral	*C*	::=	`class` *id*`@(A){` $\overline{Channel}$ $\overline{Field}$ $\overline{Method}$ `}`
Type Expression	*Type*	::=	*id*`@(A)`
Channels	*Channel*	::=	`DiChannel@(A, B) ch_A_B`
Field	*Field*	::=	*Type id*
Method Definition	*Method*	::=	*Type id*`(`$\overline{Type\ id}$`){ return` *Exp* `}`
Expression	*Exp*	::=	*id* | *Exp.id* | *Exp.id*`(`$\overline{Exp}$`)` | `new` *id*`@(A)(`$\overline{Exp}$`)`
		|	*lit*`@(A)` | `if (`*Exp*`) {` *Exp* `} else {`*Exp*`}` | *Exp*; *Exp*
		|	*ch_A_B*`.com(`*Exp*`)` | *ch_A_B*`.select(`*Exp*`)`

In Mini Choral, we decided to focus on describing *data flow* and to limit Choral's expressivity regarding *data distribution*. That is, we allow only the declared `class` to be distributed at multiple roles, while variables belong to only one role, with the exception of *Channel*s, which specify the network topology as a set of objects located (and able to transfer single-role objects) between two roles. Specifically, Mini Choral supports only one-way channels (drawn from Choral and called `DiChannels`) of the shape `DiChannel@(A,B) ch_A_B` – with `A` and `B` roles of the enclosing class. In this work, the loss of expressiveness of the Mini variant with respect to Choral – which supports the definition of multi-role classes/fields without the above limitations – lends itself to simplify the algorithms in our translation in Section 5. In the general case, Choral can express arbitrary channel topologies and user-defined implementations of communications semantics (e.g., asymmetric channels or bidirectional symmetric channels) [16] – whereas most choreographic languages assume a complete topology of channels between all roles in a choreography with a fixed communication semantics [4, 11].

Following the considerations above, we restrict type expressions *Type* to define variables located at one role *id*`@(A)`. This is reflected in the definition of *Field*s but also in method definitions, where we additionally assume the return type *Type* and the types of arguments $\overline{Type\ id}$ to be located at the same role. The body of the method is the single statement `return` *Exp*. Regarding expressions, we focus our description on the relevant, non-standard elements: object creation `new` *id*`@(A)(`$\overline{Exp}$`)` happens for classes at only one role and literals *lit*`@(A)` (integers, strings, etc.) are always located at one role. In *Exp*, we use *Exp*; *Exp* to represent a block which evaluates the expression on the left, discards its value, and returns the evaluation of the expression on the right.

Although already captured by the grammar, we include channel invocations of the shape *ch_A_B*`.com(`*Exp*`)` and *ch_A_B*`.select(`*Exp*`)` to highlight their relevance in the language. `DiChannels` support both methods `com`, meant to transfer data between two roles, and `select`, used to solve knowledge-of-choice challenges in conditionals (that is, informing a role of a local choice made by another role, e.g., by using a conditional) [16]. When using `selects`, we assume that the compiler provides us with a `Choice enum` class at one role, with a `THEN` and `ELSE` inhabitants (as presented at Line 1 in Listing 2).

4.1.1 Example: Mini Choral Expressiveness

We conclude the presentation of our minimal choreographic language by illustrating its expressiveness with respect to its reference Choral language with an implementation of the email-fetching protocol presented in Section 3.2, Listing 2.

We report the code of the Mini Choral implementation of the protocol in Figure 3 in Listing 5. In the Listing, the main notable difference with Listing 2 is that, by removing assignments, we rely on method bindings to reuse variables in "subsequent" (;) invocations.

Listing 5 Mini Choral implementation for the context-aware email fetching example.

```
class EmailSystem@(Client, Server) {
  DiChannel@(Client, Server) ch_Client_Server
  DiChannel@(Server, Client) ch_Server_Client

  MailServerDB@Server serverDB
  MailDB@Client clientDB

  Unit@Client updateEmails(UserId@Client userId) {
    return contextAwareUpdate(getEmails(userID, clientDB.lastCheckOut()))
  }

  List@Client getEmails(UserId@Client id, Timestamp@Client ts) {
    return ch_Server_Client.com(
      serverDB.since(ch_Client_Server.com(id), ch_Client_Server.com(ts))
  }

  Unit@Client contextAwareUpdate(List@Client emails) {
    clientDB.update(emails);
      if (ClientLib.isOnFlatRate()) {
        ch_Client_Server.select(Choice@Client.THEN);
        clientDB.updateAttachments(
          ch_Server_Client.com(
            serverDB.getAttachments(
              ch_Client_Server.com(clientDB.extractIds(emails)))))
      }
      else {
        ch_Client_Server.select(Choice@Client.ELSE); Unit
      }
  }
}
```

Although divided into three sub-methods, we find the `updateEmails` method that invokes the `getEmails` method, which fetches the emails from the `Server` by sending to it the `id` of the user and the timestamp (`ts`) of the last checkout and transmitting back the result of the extraction on the `serverDB`. Notice that the return type of the `getEmails` method omits the definition of the "content" of the list due to the lack of generics. As expected, by omitting generics we also drop support for specifying/checking the correct/expected content of the collection – an orthogonal guarantee with respect to the specification/check of the flow of data among roles. The lack of generics does not hamper the expressiveness of the language to capture the correct movement of the data from the `Server` to the `Client` and vice versa. After obtaining the emails, we can apply method `contextAwareUpdate` which updates the email database of the client and proceeds to conditionally retrieve the attachments of the fetched emails. This is done by informing the `Server` of the choice, via the `select` methods.

4.2 Mini ScalaLoci

Listing 6 displays the grammar of Mini ScalaLoci. L ranges over object declarations, *Peer* ranges over peer declarations, *Field* ranges over class fields, *Method* ranges over method definitions, *Type* ranges over type expressions, *PlacedType* ranges over placement type expressions, *Exp* ranges over expressions, and *PlacedExp* ranges over placed expressions. The metavariable *id* ranges over both class names, fields, and variables. We use `A`, `B`, `C` to range over peers.

The object declaration L defines its name *id*, and its peers $\overline{A}$ and topology of directed ties between the peers within the `@peer type` A `<: { type Tie <: Any with Single[`A`] }` clauses, its field declarations $\overline{Field}$, and its method definitions $\overline{Method}$. Fields associate a placement type expression *PlacedType* to a variable.

Listing 6 Syntax of Mini ScalaLoci.

Mini ScalaLoci	L	::=	`@multitier object` { $\overline{Peer}$ $\overline{Field}$ $\overline{Method}$ }
Peer	$Peer$	::=	`@peer type` A `<:` { `type Tie <: Any with Single[B]` }
Placement Type Expression	$PlacedType$	::=	$Type$ `on` A
Type Expression	$Type$	::=	id
Field	$Field$	::=	`val` id : $PlacedType$
Method Definition	$Method$	::=	`def` id ($\overline{id : Type}$) : $PlacedType$ = $PlacedExp$
Placed Expression	$PlacedExp$	::=	`on`[A] { Exp }
Expression	Exp	::=	id | $Exp.id$ | $Exp.id(\overline{Exp})$ | `new` $id(\overline{Exp})$
		|	lit | `if` (Exp) { Exp } `else` { Exp } | Exp; Exp
		|	`on`[A]`.run.capture`($\overline{id}$) { Exp }`.asLocal`

Mini ScalaLoci is able to express different *topologies* rather than being restricted to *a fixed client-server* model. This choice remarks the departure taken by ScalaLoci from other multitier models and implementations [8, 9, 34, 35, 36], which assume a fixed client-server or n-tier architecture of an application. Contrarily, in ScalaLoci, the developer defines an arbitrary number of peers and directional ties between them. In contrast to ScalaLoci, Mini ScalaLoci only supports a single connected peer instance per peer type (drawn from ScalaLoci's `Single` ties) of the shape `@peer type A <: { type Tie <: Any with Single[A] }` – with `A` and `B` peers of the enclosing multitier module. (In Scala, `with` is the operator for constructing intersection types.) In this work, the loss of expressiveness of the Mini variant with respect to ScalaLoci lends itself to simplify the algorithms in our translation in Section 5.

In method definitions, the return type *PlacedType* specifies a location, which places the computation of the whole method on that peer, whereas the arguments only have types but no placement $\overline{id : Type}$. The body of the method is a placed expression *PlacedExp* that specifies the placement of the contained expression *Exp*. Regarding expressions, we focus our description on the main differences with Choral: In ScalaLoci, we locate expressions rather than data and therefore neither instantiation `new` $id(\overline{Exp})$ nor literals lit (integers, strings, etc.) carry placement annotations.

Nested remote blocks are encoded by `on`[A]`.run.capture`($\overline{id}$) { Exp }`.asLocal` expressions, which execute the nested expression on the peer A and returns its result via `asLocal` to the surrounding peer, i.e., switching the current perspective to another peer for evaluating the nested expression. Note that in the Mini variant, we keep the `run`, `capture` and `asLocal` constructs to be close to the complete version of the ScalaLoci language (that is syntactically more flexible and supports optional `capture` clauses and `asLocal` on module-level value bindings).

4.2.1 Example: Mini ScalaLoci Expressiveness

We show the implementation of the email-fetching example presented in Section 3.3, Listing 3 using our minimal multitier language to demonstrate its expressiveness with respect to its reference ScalaLoci language.

Listing 7 shows the Mini ScalaLoci implementation of the communication scheme in Figure 3. As with Mini Choral, the main notable difference with Listing 3 is that by removing assignments, we rely on method arguments for scoped variable declarations instead. The `updateEmails` method invokes the `getEmails` method, which fetches the emails from the `Server` by sending to it the `id` of the user and the timestamp (`ts`) of the last checkout and transmitting back the result of the extraction on the `serverDB`. Similar to Mini Choral, Mini ScalaLoci also lacks generics, an orthogonal language feature. The lack of generics, however, does not limit the expressiveness of the language to capture the correct topology of the system and

Listing 7 Mini ScalaLoci implementation for the context-aware email fetching example.

```
@multitier object EmailSystem {
  @peer type Client <: { type Tie <: Any with Single[Server] }
  @peer type Server <: { type Tie <: Any with Single[Client] }

  val serverDB: MailServerDB on Server
  val clientDB: MailDB on Client

  def updateEmails(userId: UserId): Unit on Client = on[Client] {
    contextAwareUpdate(getEmails(userId, clientDB.lastCheckOut()))
  }

  def getEmails(id: UserId, ts: Timestamp): List on Client = on[Client] {
    on[Server].run.capture(id, ts) { serverDB.since(id,ts) }.asLocal
  }

  def contextAwareUpdate(emails: List): Unit on Client = on[Client] {
    clientDB.update(emails);
    if (ClientLib.isOnFlatRate()) {
      updateAttachments(clientDB.extractIds(emails))
    }
    else { () }
  }

  def updateAttachments(emailIds: List): Unit on Client = on[Client] {
    clientDB.updateAttachments(
      on[Server].run.capture(emailIds){
        serverDB.getAttachments(emailIds)
      }.asLocal
    )
  }
}
```

communication between the `Server` and the `Client`. After obtaining the emails, we apply method `contextAwareUpdate`, which updates the email database of the client and proceeds to conditionally retrieve the attachments of the fetched emails.

5 Choreographies to Multitier, Multitier to Choreographies

We now define algorithms that translate programs in a Mini language to the other and vice versa. The reason for defining the following algorithms is to present evidence of the existence of a common root at the foundation of the two approaches. We show that the mechanised procedures for their reciprocal translation are relatively simple. In the remainder of this section, for brevity, we use the names Choral and ScalaLoci to indicate their Mini counterparts. We first present a translation algorithm from a Choral choreography to a ScalaLoci multitier application (Section 5.1). Afterwards, we show a translation algorithm from a ScalaLoci multitier application to a Choral choreography (Section 5.2).

Perspective translation. Multitier and choreographic programming take different perspectives on what parts of the language are annotated with locations. In Choral, all literals are annotated by the role on which they operate, and the location of operators can be inferred by the location of their argument. ScalaLoci assigns peers to expressions, which are then written from the specified peer's perspective.

While in simple cases there is a direct correspondence between a value on the role `A` in Choral (`1@A`) and on a peer `A` in ScalaLoci (`on[A] { 1 }`), the difference is more obvious in compound expressions (`on[A] { 1 + 2 + 3 }` vs. `1@A + 2@A + 3@A`), where in ScalaLoci, only the

Algorithm 1 Translation algorithm from Choral classes to ScalaLoci objects.

function *choral2loci(class)*
  "`class` id`@`$(\overline{Role})$ { $\overline{Channel}$ $\overline{Field}$ $\overline{Method}$ }" ← class
  $decls$ ← { }
  for T ← $Role$ **do**
    $ties$ ← { "`Single`[B]" | "DiChannel@(A, B) ch_A_B" ∈ $Channel$ ∧ $T = A$ }
    $decls$ ← $decls$ ∪ { "`@peer type` T `<:` { `type Tie <: Any with` $\overline{ties}$ }" }
  end
  for "id_t`@`(A) id_n" ← $\overline{Field}$ **do**
    $decls$ ← $decls$ ∪ { "`val` id_1 : id_0 `on` A" }
  end
  for "id_t`@`(A) id $(\overline{id_{tn}\texttt{@}(A)\ id_{en}})$ { e }" ← $\overline{Method}$ **do**
    e' ← $choral2loci(e)$
    $decls$ ← $decls$ ∪ { "`def` $id(\overline{id_{en} : id_{tn}})$: id_t `on` $A = \{e'\}$" }
  end
  return "`@multitier object` id { $\overline{decls}$ }"
end

whole expression is annotated but the literals are not, whereas in Choral, only the literals are annotated while the expression is not.

The translation algorithms perform such perspective change by grouping composed literals on the same Choral role into a ScalaLoci placed expression and, in the opposite direction, assigning the same Choral role to all literals in a ScalaLoci placed expression.

Further, we translate between ScalaLoci's way of defining peers and their topology as type members and Choral's way of defining roles as class parameters and their communication channels as class members.

Communication translation. In ScalaLoci two peers communicate using `asLocal`. Given an expression `e` on peer `A`, the expression `on[B] { e.asLocal }` describes how peer `B` can access the value of `e`, implemented as a message with the value of `e` sent from `A` to `B`. In Choral, such communication is represented by invoking the `com` method of a directional communication channel, which takes a value on role `A` and returns it on role `B`.

The translation algorithms transform `asLocal` in ScalaLoci to an invocation of method `com` of the appropriate channel in Choral and vice versa.

5.1 From Choreographic Programming to Multitier Programming

Choral choreography classes to ScalaLoci multitier objects. Algorithm 1 describes the translation of Choral choreography classes to ScalaLoci multitier objects. We decompose the class definition to be transformed into its identifier id, the roles $\overline{Role}$, the channel declarations $\overline{Channel}$, the field declarations $\overline{Field}$ and the method definitions $\overline{Method}$.

Each Choral role definition is translated to a ScalaLoci peer definition. Each channel `DiChannel@(A,B) ch_A_B` between two roles is translated to a single tie, e.g., a directed one-to-one tie, between two peers `@peer type A <: { type Tie <: Any with Single[B] }`.

The translation of field definitions from Choral to ScalaLoci is straightforward. In Choral, fields are introduced with a base type and the residing role, followed by the name of the field "id_{name}`@`(id_{role}) id_{type}". In ScalaLoci, fields are introduced as "`val` id_{name}: id_{type} `on` id_{role}". Similarly, method definitions are translated. The algorithm returns a multitier object with the same name and the translated definitions as a body.

Algorithm 2 Translation algorithm from Choral expressions to ScalaLoci expressions.

```
function choral2loci(expr)
    return match expr with
        case "e0; e1" with
            "on[A]{ e0' }" ← choral2loci(e0)
            "on[B]{ e1' }" ← choral2loci(e1)
            captures ← freeVars(e0) ∩ currentMethodArguments
            if A ≠ B then
                "on[B]{ on[A].run.capture(captures) { e0' }.asLocal; e1' }"
            else
                "on[B]{ e0'; e1' }"
            end
        case "id" with
            A ← roleOf(id)
            "on[A]{ id }"
        case "lit@A" with
            "on[A]{ lit }"
        case "new id@A (ē)" with
            "on[A]{ e' }" ← choral2loci(e)
            "on[A]{ new id(ē') }"
        case "e0.id(ē)" with
            "on[A]{ e0' }" ← choral2loci(e0)
            "on[B]{ e' }" ← choral2loci(e)
            assert A = B // receiver and arguments have the same role
            "on[A]{ e0'.id(ē') }"
        case "ch.select(e)" with
            "Unit"
        case "if (e0) { e1 } else { e2 }" with
            "on[A]{ e0' }" ← choral2loci(e0)
            "on[B]{ e1' }" ← choral2loci(e1)
            "on[C]{ e2' }" ← choral2loci(e2)
            captures ← freeVars(e0) ∩ currentMethodArguments
            assert B = C // branches have the same role
            if A ≠ B then
                "on[B]{ if (on[A].run.capture(captures) { e0' }.asLocal) { e1' } else { e2' } }"
            else
                "on[B]{ if (e0') { e1' } else { e2' } }"
            end
        case "ch_B_A.com(e)" with
            "on[B]{ e' }" ← choral2loci(e)
            captures ← freeVars(e) ∩ currentMethodArguments
            "on[A]{ on[B].run.capture(captures) { e' }.asLocal }"
    end
end
```

Choral choreography expressions to ScalaLoci multitier expressions. Algorithm 2 describes the translation of Choral expressions to ScalaLoci: the algorithm matches on the different cases of Choral *Exp* terms and transforms each into the corresponding ScalaLoci code.

For sequencing $e_0; e_1$, both e_0 and e_1 are recursively transformed. If both subexpressions agree on their placement, e.g., $A = B$, the complete sequence is placed on the same peer. More interestingly, if the subexpressions are placed on different peers, we introduce a nested remote block for e_0', which executes e_0' on A and places the overall result of e_1' on B. For the remote block we generate a capture clause for all method-local variables that are free in e_0.

Algorithm 3 Translation algorithm from ScalaLoci objects to Choral classes.

```
function loci2choral(module)
    "@multitier object id { Peer* Field* Method* }" ← module
    decls ← { }
    roles ← { }
    for "@peer type A <: { type Tie <: Any with ties* }" ← Peer do
        roles ← roles ∪ { A }
        for "Single[B]" ← ties do
            decls ← decls ∪ { "DiChannel@(A,B) ch_A_B" }
        end
    end
    for "val id1: id0 on A" ← Field do
        decls ← decls ∪ { "id0@(A) id1" }
    end
    for "def id(id_en : id_tn): id_t on A = { e }" ← Method do
        e' ← loci2choral(e)
        decls ← decls ∪ { "id_t@(A) id(id_tn@(A) id_en) { e' }" }
    end
    return "class id@(roles*) { decls* }"
end
```

The translations for identifiers, literals and instantiation is straightforward, placing the ScalaLoci expression on the peer according to the role specified in the Choral code. Further, the case for method invocation is similar since we assume that the receiver of a method invocation and its arguments are on the same role. This assumption is expressed by the *assert* statement in the algorithm and holds for every well-typed Mini Choral program (in contrast to a Choral program). Selection does not exist in ScalaLoci. Hence, it is removed.

The case for branching makes a distinction similar to sequencing of whether the condition agrees to the branches regarding their placement, e.g., $A = B$. If they agree, the complete branching is placed on the same peer. Otherwise, we introduce a nested remote block for e'_0, which executes e'_0 on A and returns the result to B where the branches are placed. B then acts as a coordinator to decide which of the branches to execute.

Finally, we translate Choral's channel communication. For a channel from role B to A, we generate a ScalaLoci expression, which runs a nested remote block for e', which executes e' on B and returns the result to A.

5.2 From Multitier Programming to Choreographic Programming

ScalaLoci multitier objects to Choral choreography classes. Algorithm 3 describes the translation of ScalaLoci multitier objects to Choral choreography classes. We decompose the multitier object to be transformed into its identifier *id*, the peer and tie declarations $\overline{Peer}$, the field declarations $\overline{Field}$ and the method definitions $\overline{Method}$.

Each ScalaLoci peer definition is translated to Choral role argument and each single tie between two peers is translated to a `DiChannel` between two peers `@(A,B)`.

The translation of fields and methods from ScalaLoci to Choral is straightforward. The algorithm returns a Choral class with the same name and the translated definitions as body.

ScalaLoci multitier expressions to Choral choreography expressions. Algorithm 4 describes the translation of ScalaLoci expressions to Choral expressions. The algorithm matches on the different cases of ScalaLoci *Expr* terms and transforms each of them into the corresponding ScalaLoci code.

Algorithm 4 Translation algorithm from ScalaLoci expressions to Choral expressions.

function *loci2choral(expr)*
  return match *expr* **with**
    case "**on**[A]{ e_0; e_1 }" **with**
      $e'_0 \leftarrow$ *loci2choral*("**on**[A]{ e_0 }")
      $e'_1 \leftarrow$ *loci2choral*("**on**[A]{ e_1 }")
      "e'_0; e'_1"
    case "**on**[A]{ id }" **with** "id"
    case "**on**[A]{ lit }" **with**
      "lit@A"
    case "**on**[A]{ **new** $id(\overline{e})$ }" **with**
      $\overline{e' \leftarrow \textit{loci2choral}(\text{"}\textbf{on}[A]\{\ e\ \}\text{"})}$
      "**new** id@$A(\overline{e'})$"
    case "**on**[A]{ $e_0.id(\overline{e})$ }" **with**
      $e'_0 \leftarrow$ *loci2choral*("**on**[A]{ e_0 }")
      $\overline{e' \leftarrow \textit{loci2choral}(\text{"}\textbf{on}[A]\{\ e\ \}\text{"})}$
      "$e'_0.id(\overline{e'})$"
    case "**on**[A] { **if** (e_0) { e_1 } **else** { e_2 } }" **with**
      $e'_0 \leftarrow$ *loci2choral*("**on**[A]{ e_0 }")
      $e'_1 \leftarrow$ *loci2choral*("**on**[A]{ e_1 }")
      $e'_2 \leftarrow$ *loci2choral*("**on**[A]{ e_2 }")
      $peers \leftarrow peersIn(e'_1) \cup peersIn(e'_2)$
      $channels \leftarrow$ { "ch_A_B" | $B \in peers \wedge A$ has tie to B }
      $thenSelects \leftarrow$ { "c.*select*(Choice@A.THEN)" | $c \in channels$ }
      $elseSelects \leftarrow$ { "c.*select*(Choice@A.ELSE)" | $c \in channels$ }
      "**if** (e'_0) { $\overline{thenSelects}$; e'_1 } **else** { $\overline{elseSelects}$; e'_2 }"
    case "**on**[A]{ **on**[B].**run**.**capture**($\overline{captures}$) { e }.**asLocal** }" **with**
      $e' \leftarrow$ *loci2choral*("**on**[B]{e}")
      "ch_B_A.com(e')"
  end
end

The translations for sequencing, identifiers, literals, instantiation and method invocation is straightforward, recursively transforming each subexpression.

In the case for branching, the translation needs to synthesise select expressions to implement knowledge of choice (recall Section 3.2). Hence, we collect all peers used in the branches and create select statements for all channels between those peers for both branches.

Finally, we translate ScalaLoci's nested remote blocks. For a remote expression placed on A that executes e on B, we generate a Choral channel communication that transfers the value of e from B to A.

6 A Unified Perspective

Although choreographic and multitier programming evolved in dissimilar ways, their cores – represented by our two Mini languages – are close enough to let us define in Section 5 straightforward translation algorithms in both directions and show the core features of both approaches isomorphic.

Besides the more abstract purpose to present evidence of the closeness of the two approaches, our translation algorithms are also directly useful in practice. Translating Choral to ScalaLoci code enables the reuse of ScalaLoci's middleware for Choral. In general, translating to multitier programs is interesting because we can leverage the possibility of compiling to different technologies.

Table 1 Overview of the feature comparison of choreographic and multitier programming.

Feature	Choral	ScalaLoci
Distributed Data Structures (Section 6.1.1)	✓	×
Dynamic Topologies (Section 6.1.2)	×	✓
Higher-Order Composition (Section 6.1.3)	✓	×
Races (Section 6.1.4)	−	✓
Fault tolerance (Section 6.1.5)	✓	✓
Asynchrony (Section 6.1.6)	✓	✓

Translating ScalaLoci to Choral code enables synthesising the choreography of the multitier program. Making the protocol manifest supports both manually checking what communications take place as well as automatic analyses (e.g., security).

We believe that both the multitier and choreographic research areas can greatly benefit from cross-fertilisation and transfer of concepts already developed in one but lacking in the other. As a glimpse of this fact, we dedicate Section 6.1 to describe some advanced features present in only one of the two languages (Choral, ScalaLoci) and outline how they could be integrated into the other in the future. We conclude this section by widening our scope on the category of multiparty language in Section 6.2. We give an (incomplete) overview on other languages that are neither multitier nor choreographic but share common traits that can classify them as multiparty ones. We consider those languages valuable additions to the multiparty category and subject of future research akin to this work.

6.1 Feature Comparison

We now discuss a few features that are important for concurrent and distributed programming. Our discussion is summarised in Table 1, which shows which features are present in Choral and ScalaLoci, respectively (the − in the table means partial support, explained in the relevant paragraph where we discuss the feature). The first four features have evolved separately and give potential for cross-fertilisation, whereas the last two are important features that have been dealt in both worlds (yet separately).

6.1.1 Distributed Data Structures

The `@(R1, ..., Rn)` type notation supported in Choral specifies the distribution of classes and objects over roles. This is true also without taking into account communication. As an example, let us consider the `BiPair` class below, which implements an incarnation of a `Pair` class where the two values (referred to as `left` and `right`) of the pair belong to different roles:

```
class BiPair@(A,B)<L@X, R@Y> {
  private L@A left;
  private R@B right;
  public BiPair(L@A left, R@B right) { this.left = left; this.right = right; }
  public L@A left() { return this.left; }
  public R@A right() { return this.right; }
}
```

As its Java counterpart, also `BiPair` is parametric with respect to its contents: we use parameters `L` and `R` to capture the type of the left and right components of the pair. Then, by specifying that `L` is owned by one role `X` and `R` is owned by another role `Y`, we indicate that

the two values in the pair must be at different roles (and they can capture different data types, e.g., `String` and `Integer`). Indeed, adopting the same interpretation of Java generics, Choral interprets role parameter binders so that the first appearance of a parameter is a binder, while subsequent appearances of the same parameter are bound – hence, given that the declaration of type parameters `<...>` limits the scope of the of role parameters `X` and `Y`, we are indicating that they cannot coincide. For completeness, we include in the definition of the `BiPair` class its fields (`left` and `right`, respectively located at `A` and `B`), a constructor, and the traditional accessors.

Besides showing the basic feature of inherent distribution supported by the Choral type system, the example of `BiPair` is useful to illustrate that, also without considering communications, Choral offers support in defining programs where the data at some role needs to correlate with data at another, e.g., as in the case of distributed authentication tokens.

Similar to Choral, in ScalaLoci, we use parameters `L` and `R` to capture the type of the left and right components of the pair. Corresponding to Choral's roles definition, we define an `A` and a `B` peer type. We then specify that `L` is placed on a peer `A` and `R` is placed on a peer `B`:

```
@multitier trait BiPair[L, R] {
  @peer type A
  @peer type B

  val left: L on A
  val right: R on B
}
```

While we can define distributed data structures similar to Choral, their usability is more limited: they need to be composed at compile-time, because of ScalaLoci's lack of higher-order composition (see Section 6.1.3).

6.1.2 Dynamic Topologies and Homogenous Behaviours

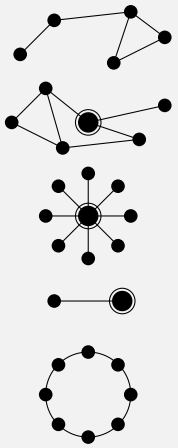
A feature of ScalaLoci that is not covered in its Mini variant is the possibility for peer types to abstract over multiple peer instances of the same type, e.g., a master-worker architecture where a single master can connect to an arbitrary number of homogeneous (i.e., with the same behaviour) worker nodes. Such a feature also enables dynamic topologies where peers can join and leave the system at runtime. A variable number of peer instances is expressed in ScalaLoci's peer specification by not using a `Single` tie but a `Multiple` or an `Optional` tie, i.e., an arbitrary number or at most one remote peer of a given type can connect, respectively.

Listing 8 shows the definitions for different topologies with their iconification on the right. The `P2P` module defines a `Peer` that can connect to arbitrary many other peers. The `P2PRegistry` module adds a central registry, to which peers can connect. The `MultiClient-Server` module defines a client that is always connected to a single server, while the server can handle multiple clients simultaneously. The `ClientServer` module specifies a server that always handles a single client instance. For the `Ring` module, we define a `Prev` and a `Next` peer. A `RingNode` itself is both a predecessor and a successor. All `Node` peers have a single tie to their predecessor and a single tie to their successor.

ScalaLoci allows to abstract over different peer instances of the same type and uniformly receive values from multiple connected remote peers, `asLocalFromAll` returns a sequence that contains the remote values from the different peers. Yet, a specific peer instance `client` can be selected via `on(client).run { ... }.asLocal` (using the `client` value referencing a peer

Listing 8 Distributed Architectures.

```
@multitier object P2P {
  @peer type Peer <: { type Tie <: Multiple[Peer] }
}
@multitier object P2PRegistry {
  @peer type Registry <: { type Tie <: Multiple[Peer] }
  @peer type Peer <: { type Tie <: Optional[Registry] with Multiple[Peer] }
}
@multitier object MultiClientServer {
  @peer type Server <: { type Tie <: Multiple[Client] }
  @peer type Client <: { type Tie <: Single[Server] with Single[Node] }
}
@multitier object ClientServer {
  @peer type Server <: { type Tie <: Single[Client] }
  @peer type Client <: { type Tie <: Single[Server] with Single[Node] }
}
@multitier object Ring {
  @peer type Node <: { type Tie <: Single[Prev] with Single[Next] }
  @peer type Prev <: Node
  @peer type Next <: Node
  @peer type RingNode <: Prev with Next
}
```



instance) instead of `on[Client].run { ... }.asLocal` (using the `Client` peer type). The handlers `remote[Client].join foreach { ... }` and `remote[Client].leave foreach { ... }` can be used to react to dynamic changes in the topology of the running multitier system.

Denièlou and Yoshida [13] developed a theory for choreographies with homogeneous roles and dynamic topologies by allowing choreographies to be parametrised (also) in collections of roles. Plans for supporting for this feature in Choral are discussed in [16, §7]. In this extension, prefixing a role parameter declaration with $*$, as in `*Clients`, specifies that this is a collection of roles. Types are extended with products indexed over collections of role using a syntax similar to Java `for-each` blocks. For instance, the type `forall(Client: Clients) String@Client` represents a "tuple" with a `String` for each role in the collection `Clients`. We can write a scatter-gather channel over a star topology (cf. `MultiClientServer`) as follows:

```
abstract class StarChannel@(Server, *Clients) {
  forall (Client : Clients) { SymChannel@(Server,Client) } star;
  forall (Client : Clients) { String@Client } scatter(String@Server m);
  String@Server gather(forall (Client : Clients) { String@Client } ms);
}
```

Method `gather` of `StarChannel` is then translated to ScalaLoci's primitive `asLocalFromAll` and vice versa. A further extension discussed in [16, §7] is the introduction of existential quantification over roles in role collections. For instance, `with(Client: Clients) {String@(Client)}` represents a string at some role in the collection `Clients`. We can extend the example above to support any-cast communication as follows:

```
abstract class StarChannel@(Server, *Clients) {
  /* ... */
  with (Client : Clients) { String@(Client) } any(String@Server m);
  String@Server any(with (Client : Clients) { String@(Client) } m);
}
```

Method `any` of `StarChannel` is then translated to ScalaLoci's `on(c).run { ... }` and vice versa.

6.1.3 Higher-Order and First-Class Multiparty Programs

We classify "higher-order" a multiparty language where multiparty components (objects, functions) are values that can be passed as arguments.

Choral is higher-order because methods can accept choreographic objects with multiple roles as parameters. In Choral, `Channels` are one of the most basic examples of the usage of the higher-order feature. For example, we can pass a `DiChannel` as an argument:

```
class MyClass@(A, B){
  void passValue(DiChannel@(A, B) ch) {
    ch.com<Integer>(5@B);
  }
}
```

In the example, the method `passValue` takes as input the choreographic object `DiChannel` and, by invoking its `com` method, we execute the protocol needed to send the data (`5@B`) between the two roles.

ScalaLoci does not support higher-order composition (no multitier objects as values or dynamic multitier object storage) but at least supports statically-composed modules [39]. The following snippet shows the declaration of a `ClientServer` multitier module that is parameterised over a `Client` and a `Server` peer. The module uses the monitoring functionality provided by the `Monitoring` multitier module, which is parameterised over a `Monitor` and a `Monitored` peer. The `Monitoring` module is instantiated by `mon` inside `ClientServer`. The `ClientServer` module identifies the `Client` peer with the `Monitored` peer and the `Server` peer with the `Monitor` peer and defines their ties accordingly:

```
@multitier trait Monitoring {
  @peer type Monitor { type Tie <: Single[Monitored] }
  @peer type Monitored { type Tie <: Single[Monitor] }
}

@multitier object ClientServer {
  @multitier object mon extends Monitoring

  @peer type Client <: mon.Monitored { type Tie <: Single[mon.Monitor] with Single[Server] }
  @peer type Server <: mon.Monitor { type Tie <: Single[mon.Monitored] with Single[Client] }
}
```

Porting higher-order composition from choreographic to multitier languages is an interesting challenge, because the way higher-order values are achieved in the former relies heavily on the objective view of choreographies. Whenever a value is returned in a multitier program, the subjective view of multitier languages requires that the value is located at a single place. It is thus unclear how a higher-order extension of multitier programming should be pursued.

To exemplify the challenge, consider that to return a data structure containing data from two distinct peers `A` and `B`, one of the two peers must act as coordinator and collect data from the other, e.g., by nesting `on[A]{ ... on[B]{ ... }.asLocal }`. But this would return a data structure completely located at `A`, so it does not solve the problem. Alternatively, we could add a multitier operator `par` for running code at different places simultaneously, e.g., `on[A]{ ... } par on[B]{ ... }`. The result of this expression could be a multitier pair containing data at `A` and `B` respectively. However, the only way to use this data structure would be to invoke `asLocal` on the two elements of the pair from within an `on[C]` block for some peer `C`, which would again centralise control.

6.1.4 Races

In this context, by "races" we mean well-behaved and non-deterministic first-come/first-served patterns where two or more roles "race" to communicate with a target role first (and the loser is handled correctly). We distinguish two prototypical scenarios: races among producers and races among consumers.

To program a race among multiple producers in ScalaLoci, we can simply retrieve the values from all remote producers via `asLocalFromAll` and pick the first one that becomes available via `Future.firstCompletedOf` as shown in the example below:

```
Future.firstCompletedOf(
  on[Producer].run { generateValue() }.asLocalFromAll map {
    case (producerPeerInstance, value) => value map { (producerPeerInstance, _) }
  })
```

It is not possible to program a race among multiple consumers in ScalaLoci. In general, consumer races represent unexplored territory for the multitier paradigm.

In Choral, it is possible to implement protocols with races among producers and among consumers provided their number is statically fixed. For instance, below is the type for a choreography where two producers race to send a message to a consumer:

```
interface ProducerRace@(Producer1, Producer2, Consumer) {
  Message@Consumer run(Message@Producer1 m1, Message@Producer2 m2);
}
```

The constraint that the number of roles must be statically fixed is related to the inability of Choral to capture dynamic topologies and, as discussed above, is solved by adding collections of roles to the language. In the case of consumer races, another limitation is that the Choral type system is not powerful enough express (and enforce) their presence. Consider a situation where two consumers race to receive a message from a single producer. In Choral, this protocol can implement the following interface:

```
interface ConsumerRace@(Producer, Consumer1, Consumer2) {
  BiPair@(Consumer1,Consumer2)<Optional<Message>,Optional<Message>> run(Message@Producer m);
}
```

However, the return type of `run` does not guarantee that exactly one consumer receives the message: implementations that deliver the message to both or neither respect the type. As discussed in [16, §7], we can write a precise type if we extend Choral with existential quantification over roles (recall the syntax for existentials at the end of Section 6.1.2) as shown in the example below:

```
interface ConsumerRace@(Producer, Consumer1, Consumer2) {
  with(C : [Consumer1, Consumer2]) { Message@C } run(Message@Producer m);
}
```

6.1.5 Fault Tolerance

In ScalaLoci, remote values whose computation or transmission to the local peer instance fail result in a future that is completed with a failure value. Thus, user code can detect a failed remote access and decide how to react appropriately by using library APIs. For example, failed futures can be handled using the typical operators on futures like `recover`:

```
on[Client].run { generateValue() }.asLocal recover { case _ => generateOtherValue() }
```

Similarly, Choral does not commit to specific failure handling mechanisms at the language level: programmers can implement their own strategies, e.g., returning errors. An API for

channels that is equivalent to the `recover` library method above could look as follows (from the point of view of the caller):

```
chAB.comOrRecover(generateValue(), new OtherValueGenerator@B());
```

where `OtherValueGenerator` has a `run` method equivalent to `generateOtherValue()`. Similar observations hold for timeouts.

ScalaLoci offers some APIs to trigger code when communications with peers in network with dynamic topologies timeout. If dynamic topologies are introduced to Choral, these APIs will become relevant for choreographies as well. We conjecture that they can be imported in a similar way to the one sketched above for recovery.

6.1.6 Asynchrony

For the sake of exposition, we presented multiparty programs using communication APIs as if they were blocking and designed the Mini variants of both Choral and ScalaLoci as synchronous. ScalaLoci promotes an asynchronous approach: the preferred variant of accessing remote values via `asLocal` in ScalaLoci creates a future to account for network delay and potential communication failure. On the other hand, Choral is agnostic with regards to communication models: programmers can import libraries of channels or implement their own. For instance, a communication model similar to ScalaLoci's `asLocal` is offered by the following interface:

```
1 interface AsyncDiChannel@(Sender, Receiver)<T@X> {
2   <S@Y extends T@Y> Future@Receiver<S> com(Promise@Sender<S> v);
3 }
```

6.2 Other Multiparty Languages

For the future we envision further cross-fertilisation between multiparty languages, and that the class of multiparty languages might get larger. We mention a few approaches outside of choreographic and multitier programming that might contribute to this.

Software architectures [14, 33] are about organising software systems into well-studied patterns that comprise components and their connections organised in a certain configuration. Architecture description languages (ADL) [26] specify software architectures and the constraints among the architecture components. Different from choreographic and multitier programming, ADLs usually specification languages separate from the implementation. An exception is ArchJava [1] which support specifying a software architecture and enforcing its constraints together with the implementation. Regarding cross-fertilisation, ADLs come equipped with powerful analysis, code synthesis, and runtime-support tools as well as model checkers, which can be also used in multitier and choreographic scenarios to enforce different aspects of correctness.

Partitioned global address space languages (PGAS) [12] are often used in the domain of high-performance computing. The main abstraction is a global memory address space where logical partitions are assigned to processes to maximize data locality. X10 [7] features explicit fork/join operations and provides a sophisticated dependent type system [6] to model the *place* (the heap partition) a reference points to. PGAS languages, similar to multitier and choreographic languages reduce the boundaries between hosts in a distributed system.

7 Conclusion

Choreographic and multitier languages have developed independently, leading to a number of research achievement carried out within two vibrant but separate research communities [2, 28, 40]. In this paper, we discussed the fundamental nature of the programming paradigms based on these languages, isolating the core difference between them. We then showed that, under the cover of syntactic variance, the two approaches are similar enough to be related and to reason about potential cross-fertilisation. Our observations offer a platform for future joint work between the respective communities.

References

1 Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 187–197, New York, NY, USA, 2002. ACM. doi:10.1145/581339.581365.

2 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.

3 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8:1–8:78, 2012. doi:10.1145/2220365.2220367.

4 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.

5 Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computation Structure, First International Conference, FoSSaCS'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 1998. doi:10.1007/BFb0053547.

6 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. doi:10.1145/1345206.1345211.

7 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. doi:10.1145/1094811.1094852.

8 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006. doi:10.1007/978-3-540-74792-5_12.

9 Ezra E. K. Cooper and Philip Wadler. The RPC calculus. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '09, pages 231–242, New York, NY, USA, 2009. ACM. doi:10.1145/1599410.1599439.

10 Luís Cruz-Filipe and Fabrizio Montesi. Choreographies in practice. In Elvira Albert and Ivan Lanese, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 114–123. Springer, 2016. doi:10.1007/978-3-319-39570-8_8.

11 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science*, 13(2), 2017. doi:10.23638/LMCS-13(2:1)2017.

12 Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned global address space languages. *ACM Computing Surveys*, 47(4), May 2015. doi:10.1145/2716320.

13 Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 435–446. ACM, 2011. doi:10.1145/1926385.1926435.

14 David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994. Accessed 2020-05-05. URL: http://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/intro_softarch.pdf.

15 Saverio Giallorenzo, Ivan Lanese, and Daniel Russo. Chip: A choreographic integration process. In *On the Move to Meaningful Internet Systems. OTM 2018 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part II*, pages 22–40. Springer, 2018. doi:10.1007/978-3-030-02671-4_2.

16 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choreographies as objects. *CoRR*, abs/2005.09520, 2020. arXiv:2005.09520.

17 Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular Actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI '73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. Accessed 2020-05-05. URL: http://ijcai.org/Proceedings/73/Papers/027B.pdf.

18 Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In Raja Natarajan and Adegboyega K. Ojo, editors, *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings*, volume 6536 of *Lecture Notes in Computer Science*, pages 55–75. Springer, 2011. doi:10.1007/978-3-642-19056-8_4.

19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9, 2016. Also: POPL, pages 273–284, 2008. doi:10.1145/2827695.

20 Intl. Telecommunication Union. Recommendation Z.120: Message Sequence Chart, 1996.

21 Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proc. of ASPLOS*, pages 517–530, 2016.

22 Alberto Lluch-Lafuente, Flemming Nielson, and Hanne Riis Nielson. Discretionary information flow control for interaction-oriented specifications. In *Logic, Rewriting, and Concurrency*, volume 9200 of *Lecture Notes in Computer Science*, pages 427–450. Springer, 2015.

23 Hugo A. López and Kai Heussen. Choreographing cyber-physical distributed control systems for the energy sector. In *SAC*, pages 437–443. ACM, 2017.

24 Hugo A. López, Flemming Nielson, and Hanne Riis Nielson. Enforcing availability in failure-aware communicating systems. In *FORTE*, volume 9688 of *Lecture Notes in Computer Science*, pages 195–211. Springer, 2016.

25 Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proc. of ASPLOS*, pages 329–339, 2008.

26 Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000. doi:10.1109/32.825767.

27 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. doi:10.1007/3-540-10235-3.

28 Fabrizio Montesi. *Choreographic Programming*. Ph.D. Thesis, IT University of Copenhagen, 2013. http://www.fabriziomontesi.com/files/choreographic_programming.pdf.

29 Tom Murphy VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*, pages 286–295. IEEE Computer Society, 2004. doi:10.1109/LICS.2004.1319623.

30 Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

31 Object Management Group. Business Process Model and Notation. http://www.omg.org/spec/BPMN/2.0/, 2011.

32 Peter W. O'Hearn. Experience developing and deploying concurrency analysis at facebook. In Andreas Podelski, editor, *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2018. doi:10.1007/978-3-319-99725-4_5.

33 Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992. doi:10.1145/141874.141884.

34 Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. Eliom: A core ML language for tierless web programming. In Atsushi Igarashi, editor, *Proceedings of the 14th Asian Symposium on Programming Languages and Systems*, APLAS '16, pages 377–397, Berlin, Heidelberg, 2016. Springer-Verlag. doi:10.1007/978-3-319-47958-3_20.

35 Bob Reynders, Frank Piessens, and Dominique Devriese. Gavial: Programming the web with multi-tier FRP. *The Art, Science, and Engineering of Programming*, 4(3):6:1–6:32, 2020. doi:10.22152/programming-journal.org/2020/4/6.

36 Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2.0. In Peri L. Tarr and William R. Cook, editors, *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 975–985. ACM, 2006. doi:10.1145/1176617.1176756.

37 W3C. WS Choreography Description Language. http://www.w3.org/TR/ws-cdl-10/, 2004.

38 Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with ScalaLoci. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):129:1–129:30, 2018. doi:10.1145/3276499.

39 Pascal Weisenburger and Guido Salvaneschi. Multitier modules. In Alastair F. Donaldson, editor, *Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP '19)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:29, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2019.3.

40 Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. A survey of multitier programming. *ACM Computing Surveys*, 53(4), 2020. doi:10.1145/3397495.