Mooly Sagiv (Ed.)

LNCS 3444

Programming Languages and Systems

14th European Symposium on Programming, ESOP 2005
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2005
Edinburgh, UK, April 2005, Proceedings

Springer

Lecture Notes in Computer Science 3444

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Mooly Sagiv (Ed.)

Programming Languages and Systems

14th European Symposium on Programming, ESOP 2005
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2005
Edinburgh, UK, April 4-8, 2005
Proceedings

Volume Editor

Mooly Sagiv
Tel Aviv University
School of Computer Science
Tel Aviv 69978, Israel
E-mail: msagiv@post.tau.ac.il

Library of Congress Control Number: 2005922810

CR Subject Classification (1998): D.3, D.1, D.2, F.3, F.4, E.1

ISSN 0302-9743
ISBN-10 3-540-25435-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-25435-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11410553 06/3142 5 4 3 2 1 0

Foreword

ETAPS 2005 was the eighth instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised five conferences (CC, ESOP, FASE, FOSSACS, TACAS), 17 satellite workshops (AVIS, BYTECODE, CEES, CLASE, CMSB, COCV, FAC, FESCA, FINCO, GCW-DSE, GLPL, LDTA, QAPL, SC, SLAP, TGC, UITP), seven invited lectures (not including those that were specific to the satellite events), and several tutorials. We received over 550 submissions to the five conferences this year, giving acceptance rates below 30% for each one. Congratulations to all the authors who made it to the final program! I hope that most of the other authors still found a way of participating in this exciting event and I hope you will continue submitting.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis and improvement. The languages, methodologies and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a loose confederation in which each event retains its own identity, with a separate program committee and proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronized parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for "unifying" talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

ETAPS 2005 was organized by the School of Informatics of the University of Edinburgh, in cooperation with

- European Association for Theoretical Computer Science (EATCS);
- European Association for Programming Languages and Systems (EAPLS);
- European Association of Software Science and Technology (EASST).

The organizing team comprised:

- Chair: Don Sannella
- Publicity: David Aspinall
- Satellite Events: Massimo Felici

- Secretariat: Dyane Goodchild
- Local Arrangements: Monika-Jeannette Lekuse
- Tutorials: Alberto Momigliano
- Finances: Ian Stark
- Website: Jennifer Tenzer, Daniel Winterstein
- Fundraising: Phil Wadler

ETAPS 2005 received support from the University of Edinburgh.

Overall planning for ETAPS conferences is the responsibility of its Steering Committee, whose current membership is:

> Perdita Stevens (Edinburgh, Chair), Luca Aceto (Aalborg and Reykjavík), Rastislav Bodik (Berkeley), Maura Cerioli (Genoa), Evelyn Duesterwald (IBM, USA), Hartmut Ehrig (Berlin), José Fiadeiro (Leicester), Marie-Claude Gaudel (Paris), Roberto Gorrieri (Bologna), Reiko Heckel (Paderborn), Holger Hermanns (Saarbrücken), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Kim Larsen (Aalborg), Tiziana Margaria (Dortmund), Ugo Montanari (Pisa), Hanne Riis Nielson (Copenhagen), Fernando Orejas (Barcelona), Mooly Sagiv (Tel Aviv), Don Sannella (Edinburgh), Vladimiro Sassone (Sussex), Peter Sestoft (Copenhagen), Michel Wermelinger (Lisbon), Igor Walukiewicz (Bordeaux), Andreas Zeller (Saarbrücken), Lenore Zuck (Chicago).

I would like to express my sincere gratitude to all of these people and organizations, the program committee chairs and PC members of the ETAPS conferences, the organizers of the satellite events, the speakers themselves, the many reviewers, and Springer for agreeing to publish the ETAPS proceedings. Finally, I would like to thank the organizer of ETAPS 2005, Don Sannella. He has been instrumental in the development of ETAPS since its beginning; it is quite beyond the limits of what might be expected that, in addition to all the work he has done as the original ETAPS Steering Committee Chairman and current ETAPS Treasurer, he has been prepared to take on the task of organizing this instance of ETAPS. It gives me particular pleasure to thank him for organizing ETAPS in this wonderful city of Edinburgh in this my first year as ETAPS Steering Committee Chair.

Edinburgh, January 2005

Perdita Stevens
ETAPS Steering Committee Chair

Preface

This volume contains the 29 papers presented at ESOP 2005, the 14th European Symposium on Programming, which took place in Edinburgh, UK, April 6–8, 2005. The ESOP series began in 1986 with the goal of bridging the gap between theory and practice, and the conferences continue to be devoted to explaining fundamental issues in the specification, analysis, and implementation of programming languages and systems.

The volume begins with a summary of an invited contribution by Andrew Myers titled "Programming with Explicit Security Policies," and continues with the 28 papers selected by the Program Committee from 114 submissions. Each submission was reviewed by at least three referees, and papers were selected during a 10-day electronic discussion phase.

I would like to sincerely thank the members of the Program Committee for their thorough reviews and dedicated involvement in the PC discussion. I would also like to thank the subreferees, for their diligent work. Martin Karusseit and Noam Rinetzky helped me with MetaFrame, used as the conference management software. Finally, I would like to thank Anat Lotan-Schwartz for helping me to collect the final papers and prepare these proceedings.

January 2005 Mooly Sagiv

Organization

Program Chair

Mooly Sagiv	Tel Aviv University, Israel

Program Committee

Martín Abadi	University of California at Santa Cruz, USA
Alex Aiken	Stanford University, USA
Bruno Blanchet	École Normale Supérieure, France
Luca Cardelli	Microsoft Research, UK
Patrick Cousot	École Normale Supérieure, France
Oege de Moor	Oxford University, UK
Manuel Fähndrich	Microsoft Research, USA
John Field	IBM, USA
Maurizio Gabbrielli	Università di Bologna, Italy
Chris Hankin	Imperial College London, UK
Manuel Hermenegildo	Universidad Politécnica de Madrid, Spain and University of New Mexico, USA
Xavier Leroy	INRIA Rocquencourt, France
Anders Møller	University of Aarhus, Denmark
Greg Morrisett	Harvard University, USA
David Naumann	Stevens Institute of Technology, USA
Hanne Riis Nielson	IMM, Technical University of Denmark
Peter O'Hearn	University of London, UK
Catuscia Palamidessi	INRIA Futurs Saclay and LIX, France
Thomas Reps	University of Wisconsin-Madison, USA
Martin Rinard	MIT, USA
Andrei Sabelfeld	Chalmers University and Göteborg University, Sweden
David Sangiorgi	Università di Bologna, Italy
David Schmidt	Kansas State University, USA
Scott Stoller	SUNY at Stony Brook, USA

Referees

A. Ahmed	Z. Ariola	N. Benton
E. Albert	A. Askarov	J. Berdine
A. Aldini	F. Barbanera	L. Bettini
J. Aldrich	M. Barnett	G. Bierman

D. Biernacki
C. Bodei
C. Brabrand
K. Bruce
M. Buscemi
N. Busi
B.C. Pierce
C. Calcagno
A. Cavalcanti
K. Chatzikokolakis
S.C. Mu
T. Chothia
M. Codish
A. Corradini
A. Cortesi
V. Cortiero
S. Crafa
F.D. Valenciao
O. Danvy
F. De Boer
P. Degano
G. Delzanno
D. Distefano
D. Dougherty
D. Duggan
R. Ettinger
G. File
C. Flanagan
M. Fluet
R. Focardi
C. Fourned
B. Francisco
J. Garrigue
D. Ghica
R. Giacobazzi
J.C. Godskesen
S. Goldsmith
G. Gonthier
J. Goubault-Larrecq
M.R. Hansen
J. Hickey
T. Hildebrandt
P. Hill
Y. Huenke
J. Hurd
M.J. Jaskelioff
L. Jagadeesan
A. Jeffrey
A. Kennedy
C. Kirkegaard
B. Klin
J. Kodumal
R. Komondoor
S. Krishnamurthi
B. Le Charlier
F. Levi
F. Logozzo
P. Lopez-Garcia
I. Lynagh
R. Majumdar
R. Manevich
M.C. Marinescu
A. Matos
L. Mauborgne
D. Miller
A. Miné
D. Monniaux
M. Naik
U. Neumerkel
F. Nielson
N. Nystrom
R. O'Callahan
L. Ong
L. Paolini
B. Pfitzmann
E. Poll
F. Pottier
M. Proietti
G. Puebla
S. Rajamani
A. Ravara
J. Rehof
J. Reppy
N. Rinetzky
C. Russo
D. Rémy
C. Sacerdoti Cohen
A. Sahai
A. Sasturkar
A. Schmitt
T. Schrijvers
A.S. Christensen
R. Solmi
M. Spivey
F. Spoto
T. Streicher
K. Støvring Sørensen
J.M. Talbot
T. Terauchi
L. Tesei
H. Thielecke
C. Urban
M. Vaziri
T. Veldhuizen
B. Victor
L. Vigano
J. Vouillono
Y. Wang
B. Warinschi
Y. Xie
E. Yahav
E. Zaffanella
S. Zdancewic
T. Zhao
E. Zucca

Table of Contents

Programming with Explicit Security Policies

Andrew C. Myers

Cornell University
andru@cs.cornell.edu

Abstract. Are computing systems trustworthy? To answer this, we need to know three things: what the systems are supposed to do, what they are not supposed to do, and what they actually do. All three are problematic. There is no expressive, practical way to specify what systems must do and must not do. And if we had a specification, it would likely be infeasible to show that existing computing systems satisfy it. The alternative is to design it in from the beginning: accompany programs with explicit, machine-checked security policies, written by programmers as part of program development. Trustworthy systems must safeguard the end-to-end confidentiality, integrity, and availability of information they manipulate. We currently lack both sufficiently expressive specifications for these information security properties, and sufficiently accurate methods for checking them. Fortunately there has been progress on both fronts. First, information security policies can be made more expressive than simple noninterference or access control policies, by adding notions of ownership, declassification, robustness, and erasure. Second, program analysis and transformation can be used to provide strong, automated security assurance, yielding a kind of security by construction. This is an overview of programming with explicit information security policies with an outline of some future challenges.

1 The Need for Explicit Policies

Complex computing systems now automate and integrate a constantly widening sphere of human activities. It is crucial for these systems to be trustworthy: both secure and reliable in the face of failures and malicious attacks. Yet current standard practices in software development offer weak assurance of both security and reliability. To be sure, there has been progress on automatic enforcement of simple safety properties, notably type safety. And this is valuable for protecting systems from code injection attacks such as buffer overruns. But many, perhaps most serious security risks do not rely on violating type safety. Often the exposed interface of a computing system can be used in ways unexpected by the designers. Insiders may be able to misuse the system using their privileges. Users can sometimes learn sensitive information when they should not be able to. These serious vulnerabilities are difficult to identify, analyze, and prevent.

Unfortunately, current practices for software development and verification do not seem to be on a trajectory that leads to trustworthy computing systems. Incremental progress will not lead to this goal; a different approach is needed. We have been exploring a language-based approach to building secure, trustworthy systems, in which programs are annotated with explicit, machine-checked information security policies relating to

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 1–4, 2005.

properties such as the confidentiality and integrity of information. These properties are both crucial to security and difficult to enforce. It is possible to write relatively simple *information flow* policies that usefully capture these aspects of security. These explicit policy annotations then support automatic enforcement through program analysis and transformation.

2 Limitations of Correctness

Of course, the idea of automatic verification has always been appealing—and somewhat elusive. The classic approach of verifying that programs satisfy specifications can be a powerful tool for producing reliable, correct software. However, as a way to show that programs are secure, it has some weaknesses. First, there is the well-known problem that the annotation burden is high. A second, less appreciated problem is that classic specifications with preconditions and postconditions are not enough to understand whether a program is secure. Correctness assertions abstract program behavior; if the abstraction leaves out security-relevant information, the actual program may contain security violations (especially, of confidentiality) invisible at the level of the abstraction. Thus, it's also important to understand not only what programs do but also what they *don't* do. Even if the program has no observable effect beyond what its specification describes, the specification itself may allow the confidential information to be released. A third problem is that correctness assertions don't address the possible presence of malicious users or code, which is particularly problematic for distributed systems.

3 End-to-End Information Security

If classic specification techniques are too heavyweight and yet not expressive enough, what are the alternatives? One possibility is information flow policies, which constrain how information moves through the system. For example, a policy that says some data is confidential means that the system may not let that data flow into locations where it might be viewed insecurely. This kind of policy implicitly controls the use of the data without having to name all the possible destinations, so it can be lightweight yet compatible with abstraction. Furthermore, it applies to the system as a whole, unlike access control policies, which mediate access to particular locations but do not control how information propagates. One can think of information flow policies as an application of the end-to-end principle to the problem of specifying computer security.

Information flow policies can express confidentiality and integrity properties of systems: confidentiality is about controlling where information flows to; integrity is about controlling where information flows from. Integrity is also about whether information is computed correctly, but even just an analysis of integrity as information flow is useful for ensuring that untrustworthy information is not used to update trusted information.

Fundamentally, information flow is about dependency [ABHR99], which makes sense because security cannot be understood without understanding how components depend on one another. The approach to enforcing information flow that has received the most attention is to analyze dependency at compile time using a *security type sys-*

tem [SM03]. The Jif programming language [Mye99], based on Java, is an example of a language with a type system for information security.

The other appealing aspect of information flow policies is that they can be connected to an underlying semantic security condition, noninterference. Noninterference says roughly that the low-security behavior of a system does not change when high-security inputs are changed. This condition (which has many variants) can be expressed in the context of a programming language operational semantics [VSI96], making possible a proof that a security type system constrains the behavior of the system.

4 Whole-System Security and Mutual Distrust

Many of the computing systems for which security is especially important are distributed systems serving many principals, typically distributed at least partly because of security concerns. For example, consider a web shopping service. At the least, it serves customers, who do not entirely trust the service, and the companies selling products, who do not trust the customers or each other. For this reason, the computational resources in use when a customer is shopping are located variously on the customer's computer, on the web service provider, and on the seller's computers. It is important to recognize that these principals have their own individual security requirements; the system as a whole must satisfy those requirements in order for them to participate.

To enforce information security for such a system, it is necessary to know the requirements of each of the principals. The decentralized label model [ML00] is an information flow policy language that introduces a notion of information flow policies owned by principals. For example, in the context of confidentiality, a policy $p_1 : p_2$ means that principal p_1 owns the policy and trusts principal p_2 to read the corresponding information. More generally, p_1 trusts p_2 to enforce the relevant security property on its behalf. This structure makes it possible to express a set of policies on behalf of multiple principals while keeping track of who owns (and can relax) each policy.

For example, suppose we are implementing the game of Battleship with two players, A and B. Player A wants to be able to read his own board but doesn't want B to read it, so the confidentiality label is $\{A : A\}$. For integrity, both principals want to make sure that the board is updated in accordance with the rules of the game, so the integrity label has two owned policies: $\{A : A \wedge B,\ B : A \wedge B\}$, where $A \wedge B$ is a conjunctive principal representing the fact that both A and B must trust the updates to A's board.

5 Security Through Transformation

Secure distributed systems achieve security through a variety of mechanisms, including partitioning code and data (as in the web shopping example), replication, encryption, digital signatures, access control, and capabilities. Analyzing the security of a complex system built in this fashion is currently infeasible.

Recently, we have proposed the use of automatic program transformation as a way to solve this problem [ZZNM02]. Using the security policies in a non-distributed program, the Jif/split compiler automatically partitions its code and data into a distributed system that runs securely on a collection of host machines. The hosts may be trusted to varying

degrees by the participating principals; a partitioning is secure if policies of each principal can be violated only by hosts it trusts. The transformation employs not only partitioning, but also all of the distributed security mechanisms above to generate distributed code for Jif programs. For example, given the labels above, Jif/split can split the code of a Battleship program into a secure distributed system.

6 Conclusions and Future Challenges

The ability to provably enforce end-to-end security policies with lightweight, intuitive annotations is appealing. Using policies to guide automatic transformation into a distributed system is even more powerful, giving a form of security by construction. However, research remains to be done before this approach can be put into widespread use.

Noninterference properties are too restrictive to describe the security of real-world applications. Richer notions of information security are needed: quantitative information flow, policies for limited information release, dynamic security policies [ZM04], and downgrading policies [CM04]. End-to-end analyses are also needed for other security properties, such as availability.

Checking information flow policies with a trusted compiler increases the size of the trusted computed base; techniques for certifying compilation would help.

The power of the secure program transformation technique could be extended by employing more of the tools that researchers on secure protocols have developed; secret sharing and secure function computation are obvious examples.

Strong information security requires analysis of how programs use information. Language techniques are powerful and necessary tools for solving this problem.

References

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, San Antonio, TX, January 1999.

[CM04] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proc. 11th ACM Conference on Computer and Communications Security*, October 2004.

[ML00] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[Mye99] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[SM03] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[ZM04] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *Proc. 2nd Workshop on Formal Aspects in Security and Trust*, August 2004.

[ZZNM02] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.

Trace Partitioning in Abstract Interpretation Based Static Analyzers

Laurent Mauborgne and Xavier Rival

DI, École Normale Supérieure, 45 rue d'Ulm,
75 230 Paris cedex 05, France
{Laurent.Mauborgne, Xavier.Rival}@ens.fr

Abstract. When designing a tractable static analysis, one usually needs to approximate the trace semantics. This paper proposes a systematic way of regaining some knowledge about the traces by performing the abstraction over a partition of the set of traces instead of the set itself. This systematic refinement is not only theoretical but tractable: we give automatic procedures to build pertinent partitions of the traces and show the efficiency on an implementation integrated in the Astrée static analyzer, a tool capable of dealing with industrial-size software.

1 Introduction

Usually, concrete program executions can be described with traces; yet, most static analyses abstract them and focus on proving properties of the set of reachable states. For instance, checking the absence of runtime errors in C programs can be done by computing an over-approximation of the reachable states of the program and then checking that none of these states is erroneous. When computing a set of reachable states, any information about the execution order and the concrete flow paths is lost.

However, this *reachable states abstraction* might lead to too harsh an approximation of the program behavior, resulting in a failure of the analyzer to prove the desired property. For instance, let us consider the following program:

$$\begin{array}{l}\textbf{if}(x < 0)\{\ sgn = -1;\ \}\\ \textbf{else}\{\ sgn = 1;\ \}\end{array}$$

Clearly sgn is either equal to 1 or -1 at the end of this piece of code; in particular sgn cannot be equal to 0. As a consequence, dividing by sgn is safe. However, a simple interval analysis [7] would not discover it, since the lub (least upper bound) of the intervals $[-1,-1]$ and $[1,1]$ is the interval $[-1,1]$ and $0 \in [-1,1]$. A simple fix would be to use a more expressive abstract domain. For instance, the disjunctive completion [8] of the interval domain would allow the property to be proved: an abstract value would be a finite union of intervals; hence, the analysis would report x to be in $[-1,-1] \cup [1,1]$ at the end of the above program. Yet, the cost of disjunctive completion is prohibitive. Other domains

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 5–20, 2005.

could be considered as an alternative to disjunctive completion; yet, they may also be costly in practice and their design may be involved. For instance, common relational domains like octagons [16] or polyhedra [11] would not help here, since they describe convex sets of values, so the abstract union operator is an imprecise over-approximation of the concrete union. A reduced product of the domain of intervals with a congruence domain [13] succeeds in proving the property, since -1 and 1 are both in $\{1 + 2 \times k \mid k \in \mathbb{N}\}$. However, a more intuitive way to solve the difficulty would be to relate the value of *sgn* to the way it is computed. Indeed, if the *true* branch of the conditional was executed, then $sgn = -1$; otherwise, $sgn = 1$. This amounts to keeping *some* disjunctions based on control criteria. Each element of the disjunction is related to some property about the history of concrete computations, such as "which branch of the conditional was taken". This approach was first suggested by [17]; yet, it was presented in a rather limited framework and no implementation result was provided. The same idea was already present in the context of data-flow analysis in [14] where the history of computation is traced using an automaton chosen before the analysis.

Choosing of the relevant partitioning (which explicit disjunctions to keep during the static analysis) is a rather difficult and crucial point. In practice, it can be necessary to make this choice at analysis time. Another possibility presented in [1] is to use profiling to determine the partitions, but this approach is relevant in optimization problems only.

The contribution of the paper is both theoretical and practical:

- We introduce a *theoretical framework* for trace partitioning, that can be instantiated in a broad series of cases. More partitioning configurations are supported than in [17] and the framework also supports *dynamic partitioning* (choice of the partitions during the abstract computation);
- We provide detailed practical information about the use of the trace partitioning domain. First, we describe the implementation of the domain; second, we review some strategies for partition creation during the analysis.

All the results presented in the paper are supported by the experience of the design, implementation and practical use of the Astrée static analyzer [2, 15]. This analyzer aims at certifying the absence of run-time errors (and user-defined non-desirable behaviors) in very large synchronous embedded applications such as avionics software. Trace partitioning turned out to be a very important tool to reach that goal; yet, this technique is not specific to the families of software addressed here and can be applied to almost any kind of software.

In Sect. 2, we set up a general theoretical framework for trace partitioning. The main choices for the implementation of the partitioning domain are evoked in Sect. 3; we discuss strategies for partitioning together with some practical examples in Sect. 4. Finally, we conclude in Sect. 5.

2 Theoretical Framework

This section supposes basic knowledge of the abstract interpretation framework [5]. For an introduction, the reader is referred to [9].

2.1 Definitions

Programs: We define a program P as a transition system $(\mathcal{S}, \rightarrow, \mathcal{S}_\iota)$ where $\mathcal{S}$ is the set of states of the program; $\rightarrow$ is the transition relation describing the possible execution elementary steps and $\mathcal{S}_\iota$ denotes the set of *initial states*.

Traces: We write $\mathcal{S}^\star$ for the set of all finite non-empty sequences of states. If σ is a finite sequence of states, σ_i will denote the $(i+1)^{\text{th}}$ state of the sequence, σ_0 the first state and $\sigma_\dashv$ the last state. We define $\varsigma(\sigma)$ as the set of all the states in σ. We extend this notation to sets of sequences: $\varsigma(\Sigma) \stackrel{\text{def}}{=} \bigcup_{\sigma \in \Sigma} \varsigma(\sigma)$.

If τ is a prefix of σ, we write $\tau \preceq \sigma$. A *trace* of the program P is defined as an element of $[\![P]\!] \stackrel{\text{def}}{=} \{\sigma \in \mathcal{S}^\star \mid \sigma_0 \in \mathcal{S}_\iota \wedge \forall i,\, \sigma_i \rightarrow \sigma_{i+1}\}$. Note that the set $[\![P]\!]$ is prefix-closed. An execution of the program is a possibly infinite sequence starting from an initial state and such that there is no possible transition from the final state, if any. Executions are represented by the set of their prefixes, thus avoiding the need to deal with infinite sequences.

2.2 Reachability Analysis

In order to prove safety properties about programs, one needs to approximate the set of reachable states of the programs. This is usually done in one step by the design of an abstract domain $D^\sharp$ representing sets of states and a concretization function that maps a representation of a set of states to the set of all traces containing these states only. In order to be able to refine that abstraction, we decompose it in two steps. The first step is the *reachability abstraction*, the second one the *set of states abstraction*.

We start from the most precise description of the behaviors of program P, given by the concrete semantics $[\![P]\!]$ of P, i.e the set of finite traces of P, so the concrete domain is defined as $\mathcal{P}_\preceq(\mathcal{S}^\star) \stackrel{\text{def}}{=} \{\Sigma \subseteq \mathcal{S}^\star \mid \Sigma \text{ is prefix-closed}\}$.

Reachability Abstraction: The set of reachable states of Σ can be defined by the abstraction $\alpha_R(\Sigma) \stackrel{\text{def}}{=} \{\sigma_\dashv \mid \sigma \in \Sigma\}$. Considering the concretization $\gamma_R(T) \stackrel{\text{def}}{=} \{\sigma \in \mathcal{S}^\star \mid \forall i, \sigma_i \in T\}$, we get a Galois connection $\mathcal{P}_\preceq(\mathcal{S}^\star) \underset{\alpha_R}{\overset{\gamma_R}{\leftrightarrows}} \mathcal{P}(\mathcal{S})$. This Galois connection will allow us to describe the relative precision of the refinements defined in the sequel of this section.

Set of States Abstraction: In the rest of the section, we will assume an abstract domain $D^\sharp$ representing sets of states and a concretization function[1] $\gamma : D^\sharp \rightarrow \mathcal{P}(\mathcal{S})$. Basically, $\gamma(I)$ represents the biggest set of states safely approximated by the (local) abstract invariant I. The goal of this abstraction is to compute an approximation of the set of states effectively.

2.3 Trace Discrimination

Definition 1 (Covering). *A function $\delta : E \rightarrow \mathcal{P}(F)$ is said to be a* covering *of F if and only if $\bigcup_{x \in E}(\delta(x)) = F$.*

[1] Abstract domains don't necessarily come with an abstraction function.

Definition 2 (Partition). *A function $\delta : E \to \mathcal{P}(F)$ is said to be a* partition *of F if and only if δ is a covering of F and $\forall x, y \in E,\ x \neq y \Rightarrow \delta(x) \cap \delta(y) = \emptyset$.*

Trace Discriminating Reachability Domain: Using a well-chosen function δ of $E \to \mathcal{P}(\mathcal{S}^\star)$, one can keep more information about the traces. We define the trace discriminating reachability domain D^δ_R as the set of functions from E to $\mathcal{P}(\mathcal{S})$, ordered pointwise. The trace discriminating reachability abstraction is $\alpha^\delta_R : \mathcal{P}_\preceq(\mathcal{S}^\star) \to D^\delta_R$, $\alpha^\delta_R(\Sigma)(x) \stackrel{\text{def}}{=} \{\sigma_\dashv \mid \sigma \in \Sigma \cap \delta(x)\}$. The concretization is then $\gamma^\delta_R(f) = \{\sigma \mid \forall \tau \preceq \sigma, \forall x,\ \tau \in \delta(x) \Rightarrow \tau_\dashv \in f(x)\}$ ($(\alpha^\delta_R, \gamma^\delta_R)$ form a Galois connection).

Comparing Trace Discriminating and Standard Reachability: Following [8], we compare the abstractions using the associated upper closure operators (the closure operator associated to an abstraction α, γ is $\gamma \circ \alpha$). The simple reachability upper closure maps any set of traces Σ to the set $\{\sigma \mid \forall i, \exists \tau \in \Sigma, \sigma_i = \tau_\dashv\}$ of traces composed of states in Σ. Thus, in order to give a better approximation, the new upper closure must not map any Σ to a set containing a state which was not in Σ. If δ is not a covering, then there is a sequence which is not in $\bigcup_{x \in E} \delta(x)$, and by definition of γ^δ_R, that sequence can be in any $\gamma^\delta_R(f)$, so it is very likely that D^δ_R is not as precise as the simple reachability domain. On the other hand, if $\bigcup_{x \in E} \delta(x) = \mathcal{S}^\star$, $\gamma^\delta_R \circ \alpha^\delta_R$ is always at least as precise as $\gamma_R \circ \alpha_R$.

A function $\delta : E \to \mathcal{P}(\mathcal{S}^\star)$ can distinguish a set of traces Σ_1 from a set Σ_2 if there exists x in E such that $\Sigma_1 \subseteq \delta(x)$ and $\Sigma_2 \cap \delta(x) = \emptyset$. The following theorem states that, if the covering δ can distinguish at least two executions with a state in common, then the abstraction based on δ is more precise than standard reachability. Moreover, the abstraction based on δ is always at least as precise as the standard reachability abstraction.

Theorem 1. *Let δ be a covering of $\mathcal{S}^\star$. Then, $(D^\delta_R, \gamma^\delta_R)$ is a more precise abstraction of $\mathcal{P}_\preceq(\mathcal{S}^\star)$ than $(\mathcal{S}, \gamma_R)$. Moreover, if there are two elements of $\mathcal{P}_\preceq(\mathcal{S}^\star)$ which share a state and are distinguished by δ, then the abstraction $(D^\delta_R, \gamma^\delta_R)$ of $\mathcal{P}_\preceq(\mathcal{S}^\star)$ is strictly more precise than $(\mathcal{S}, \gamma_R)$.*

Proof. By definition, $\gamma^\delta_R \circ \alpha^\delta_R(\Sigma)$ is the set of traces σ such that $\forall \tau \preceq \sigma, \forall x$, $(\tau \in \delta(x) \Rightarrow \exists \upsilon \in \Sigma \cap \delta(x),\ \tau_\dashv = \upsilon_\dashv)$. $\exists \upsilon \in \Sigma \cap \delta(x), \tau_\dashv = \upsilon_\dashv$ implies $\exists \upsilon \in \Sigma, \sigma_i = \upsilon_\dashv$. If δ is a covering, then for all τ, there is at least one x such that $\tau \in \delta(x)$. So $\gamma^\delta_R \circ \alpha^\delta_R \subseteq \gamma_R \circ \alpha_R$, meaning that the abstraction $(D^\delta_R, \gamma^\delta_R)$ of $\mathcal{P}_\preceq(\mathcal{S}^\star)$ is more precise than $(\mathcal{S}, \gamma_R)$.

To prove that we have a strictly more precise abstraction, we exhibit a set of traces Σ such that $\gamma^\delta_R \circ \alpha^\delta_R(\Sigma)$ is strictly smaller than $\gamma_R \circ \alpha_R(\Sigma)$. Following the hypothesis, let Σ_1, Σ_2, s and x be such that s is a state in $\varsigma(\Sigma_1) \cap \varsigma(\Sigma_2)$, and $\Sigma_1 \subseteq \delta(x)$ and $\Sigma_2 \cap \delta(x) = \emptyset$. Let σ be a sequence of Σ_1 such that $\sigma_\dashv = s$ (this is always possible because Σ_1 is an element of $\mathcal{P}_\preceq(\mathcal{S}^\star)$, and as such prefix-closed). Let $\Sigma = (\varsigma(\delta(x)) - \{s\})^\star \cup \Sigma_2$. Then $\varsigma(\sigma) \subseteq \varsigma(\Sigma)$, so σ is in $\gamma_R \circ \alpha_R(\Sigma)$. But whatever $\upsilon \in \Sigma \cap \delta(x)$, υ does not contain s, so it cannot end with s, hence $\sigma \notin \gamma^\delta_R \circ \alpha^\delta_R(\Sigma)$. □

Corollary 1. *If δ is a non trivial partition of $\mathcal{S}^\star$ (no $\delta(x)$ is $\mathcal{S}^\star$), then the abstraction $(D^\delta_R, \gamma^\delta_R)$ of $\mathcal{P}_\preceq(\mathcal{S}^\star)$ is strictly more precise than $(\mathcal{S}, \gamma_R)$.*

Proof. Suppose that for an x, $\forall s \in \varsigma(\delta(x))$, $\forall y \neq x, s \notin \varsigma(\delta(y))$. Then, because δ is a covering, all sequences containing a state of $\delta(x)$ is in $\delta(x)$, which means $\delta(x) = (\varsigma(\delta(x)))^\star$. Since δ is a non trivial partition of $\mathcal{S}^\star$ not all $\delta(x)$ can be of this form. So there is an x and a y such that $\delta(x)$ distinguishes between $\delta(x)$ and $\delta(y)$ having a state in common. □

In the sequel, we will consider partitions only, so the results of Theorem 1 apply.

2.4 Some Trace Partitioning Abstractions

In this paragraph, we instantiate the framework to various kinds of partitions. In this instantiation we suppose a state can be decomposed into a control state in $\mathcal{L}$ and a memory state in $\mathcal{M}$. Thus $\mathcal{S} = \mathcal{L} \times \mathcal{M}$. We also assume that the abstract domain $D^\sharp$ forgets about the control state, just keeping an approximation of the memory states.

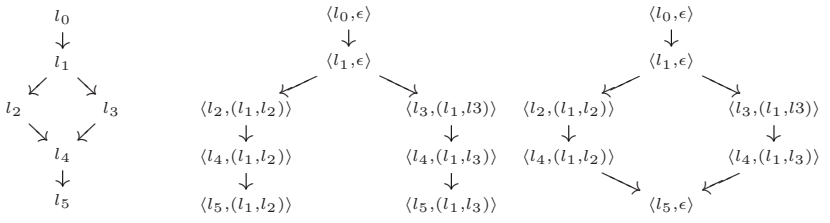
We illustrate some partitions with a simple abstract program containing a conditional on Fig 1.

Final Control State Partition: Let $\delta_\mathcal{L} : \mathcal{L} \to \mathcal{P}_\preceq(\mathcal{S}^\star)$ be the partition of $\mathcal{S}^\star$ based on the final control state: $\delta_\mathcal{L}(l) \stackrel{\text{def}}{=} \{\sigma \in \mathcal{S}^\star \mid \exists \rho,\ \sigma_\dashv = (l, \rho)\}$. This partition is very common and usually done silently when designing the abstract semantics. It leads to the abstraction $(D^\sharp_l, \gamma)$ of D, where $D^\sharp_l \stackrel{\text{def}}{=} \mathcal{L} \to D^\sharp$ and $\gamma(\mathcal{I}) \stackrel{\text{def}}{=} \{\sigma \in \mathcal{P}_\preceq(\mathcal{S}^\star) \mid \forall i,\ \sigma_i = (l_i, \rho_i) \wedge \rho_i \in \gamma(\mathcal{I}(l_i))\}$.

Control Flow Based Partition: In [17], Tzolovski and Handjieva introduced trace-based partitioning using control flow. To simplify, they proposed to extend the control states with an history of the control flow in the form of lists of tags t_i or f_i (meaning that the test number i was true or false). Then, they perform a final control state partition on this new set of control states. In order to keep the set of control states finite, they associate with each while loop an integer limiting the number of t_i to be considered.

Formally, let $\mathcal{B} \subseteq \mathcal{L}$ be the set of control points introducing a branching (e.g. conditionals, while loops...). We define $\mathbb{C} \stackrel{\text{def}}{=} \{(b, l) \in \mathcal{B} \times \mathcal{L} \mid \exists \rho, \rho' \in \mathcal{M},\ (b, \rho) \to (l, \rho')\}$ as the set of possible branch choices in the program. Note that in a branch choice (b, l), l is necessarily directly accessible from b. In order to define the trace partition used in [17], we define the control flow abstraction of a trace as the sequence $\mathrm{cf}(\sigma) \subseteq \mathbb{C}^\star$ made of the maximal sequence of branch choices taken in the trace. Then, the control flow based partition is defined as the partition $\delta_\mathrm{cf} : \mathcal{L} \times \mathbb{C}^\star \to \mathcal{P}(\mathcal{S}^\star)$, $\delta_\mathrm{cf}(l, \beta) \stackrel{\text{def}}{=} \{\sigma \in \delta_\mathcal{L}(l) \mid \mathrm{cf}(\sigma) = \beta\}$.

In order to keep the partition finite, [17] limits the number of partitions per branching control points. They use a parameter $\kappa : \mathcal{B} \to \mathbb{N}$ in the abstraction function. The κ-limiting abstraction is defined as $\lambda_\kappa(\beta)$ which is the subsequence of β obtained by deleting the branching choices $\beta_i = (b, l)$ such that if b is the conditional of a loop, the loop have been taken more than $\kappa(b)$ consecutive times



a. Control State Partition **b.** Control Flow Partition **c.** Merging Control Flow Partition

Fig. 1. Some partitions for the program $l_0 : s_0;\ l_1 : \mathbf{if}(c)\{\ l_2 : s_1;\ \}\,\mathbf{else}\,\{\ l_3 : s_2;\ \}\,l_4 : s_3;\ l_5 : s_4$

(if b is a simple branching, it is deleted if $\kappa(b)$ is 0). Then, if we use $\lambda_\kappa \circ \mathrm{cf}$ instead of cf, the effect will be to merge partitions distinguishing the simple branchings for which κ is 0 and the iterations of the loops after some point. So the partition finally used is $\delta_{\mathrm{cf}} : \mathcal{L} \times \lambda_\kappa(\mathbb{C}^\star) \to \mathcal{P}(\mathcal{S}^\star)$.

2.5 Designing Pertinent Partitions

The control flow based partition can give very precise results but is also very expensive. Even if the parameter κ is very restrictive (keeping the useful partitions only) the setting will keep the partitions after they are needed. This is very costly because each partitioning point multiplies the cost by 2 at least. Using the more general setting we describe, it is possible to define more pertinent and less expensive partitions. We describe here two examples.

Merging Control Flow: In order to reduce the cost it is possible to include in the same partition, not only the traces before a branching point, but also the traces exceeding a certain following control point. We do that if we guess that the partition based on the branching is not useful after this point. In conjunction with the final state control partition, it means that at each control point, we keep only some presumably useful partitions. Formally, we introduce a new parameter, $\mathbb{M} \subseteq \mathcal{L}$ and modify cf such that we forget everything that is before a control point in $\mathbb{M}$. To be more precise, it is even possible to use a function $\mathbb{M} \to \mathcal{B}$ and forget only the branching points corresponding to the merging point. On the example of Fig. 1, if $\mathbb{M} = \{l_5\}$, we get the partition in Fig. 1-c.

Value Based Trace Partition: The most adapted information for the definition of the partitions might not lie in the control flow. For instance, to regain some complex relationship between a variable with a few possible values and the other variables, we can add a partition according to the values of the variable at a given control point. The advantage of this approach is the very low implementation cost compared to the design of a new relational domain.

2.6 The Trace Partitioning Abstract Domain

The partitions of $\mathcal{S}^\star$ are naturally ordered by the notion of being finer (they even form a complete lattice).

Definition 3. *A partition δ_1 is finer than δ_2 if $\forall x, \exists y,\ \delta_1(x) \subseteq \delta_2(y)$. We write $\delta_1 \precsim \delta_2$.*

This lattice can be the basis of an abstract domain, using techniques inspired of the cofibered domains of [18]. The interest of such a domain is twofold. First, it allows *dynamic partitioning* by changing the partitions during the analysis (depending on the properties of the program inferred during the analysis). Second, it gives the possibility of using infinite partitions (or very big partitions) which can be abstracted away during the computation of the invariants by widening.

Basis: We introduce the notion of equivalence between partitions: δ_1 is *equivalent* to δ_2 if $\forall x, \exists y,\ \delta_1(x) = \delta_2(y)$. The *basis* of the trace partitioning abstract domain is the set $\mathfrak{T}$ of all partitions of $\mathcal{S}^\star$ up to equivalence.

Let δ_1 and δ_2 in $\mathfrak{T}$. If $\delta_1 \precsim \delta_2$, then the abstraction $D_R^{\delta_1}$ is more precise than $D_R^{\delta_2}$ if compared as closure operators, as seen in Sect. 2.3. The most precise (finest) partition distinguishes all traces: it is $\{\{\sigma\} \mid \sigma \in \mathcal{S}^\star\}$. Note that the standard reachability domain corresponds to the supremum element of $\mathfrak{T}$: we define $\delta_0 : \{0\} \to \mathcal{P}_\preceq(\mathcal{S}^\star)$ as $\delta_0(0) \stackrel{\text{def}}{=} \mathcal{S}^\star$. It is obvious that $(\mathcal{S}, \alpha_R)$ is isomorphic to $(D_R^{\delta_0}, \alpha_R^{\delta_0})$.

Definition 4 (partitioning abstract domain). *The* trace partitioning abstract domain*, $\mathbb{D}^\sharp$, is defined as: $\mathbb{D}^\sharp = \{(\delta, \mathcal{I}) \mid \delta \in \mathfrak{T} \wedge \mathcal{I} \in D_R^\delta\}$.*

Application of Dynamic Partitioning: Choosing the partitions at analysis time is crucial in many cases. For instance, the analyzer should be able to decide whether or not to operate value based partitioning (Sect. 2.5) during the analysis; indeed, in case the analysis gives no precise information about the range of an integer variable i, partitioning the traces with the values of i would lead to a dramatic analysis cost, possibly for no precision improvement. Other applications include the dynamic choice of the number of unrolled iterations in a loop (Sect. 4.4) or the analysis of recursive functions [4].

Widening: Because the basis contains infinite ascending chains, we need a widening to use the trace partitioning domain in practice. We can produce a widening on $\mathbb{D}^\sharp$ as soon as we have a widening on the basis $\nabla_\mathfrak{T}$ and a widening on the set of states abstract domain. This widening ∇ can be derived by a construction similar to [18]. To compute $(\delta_1, \mathcal{I}_1)\nabla(\delta_2, \mathcal{I}_2)$, we compute $\delta = \delta_1 \nabla_\mathfrak{T} \delta_2$ and then widen the best approximations of $\mathcal{I}_1$ and $\mathcal{I}_2$ in D_R^δ.

3 Implementation of the Domain

We now provide an overview of the data structures and algorithms which turned out the most efficient for the implementation of the trace partitioning domain.

3.1 Partition Creation and Merge

Partitions are created at *partition begin* control points. Such points are defined by *partitioning directives* The choice of the points where partition directives should be inserted will be discussed in Sect. 4.1. The main directives we implemented are: If-partitioning, Loop-partitioning, Call-stack handling, Value-partitioning. A *partition end* point merges some or all partitions. Partition begins and partition ends may or may not be well parenthesized as far as the soundness of the analysis is concerned. We may imagine some partitioning strategies that would merge older partitions first and result in more precise results. ASTRÉE assumes that partition begins and partition ends are well parenthesized for the sake of efficiency only.

A token stands for an element of the partitions observed at a (set of) control point(s). As suggested in Sect. 2.5, we partition traces according to some conditions like the choice of a branch in a conditional structure. We let such a condition be denoted by a *pre-token*. Then, a *token* is defined as the series of such conditions the execution flowed through, hence can be considered a stack of tokens. We choose a stack here instead of a list or a set, since the last partition opened should be closed first so the order of pre-tokens should be preserved.

Definition 5 (tokens). Pre-tokens *($p \in \mathcal{P}$) and* tokens *($t \in \mathcal{T}$) are defined by the following grammar (which can be extended):*

$$\begin{array}{lll} p ::= \texttt{If\_true}(l) \mid \texttt{If\_false}(l) \mid \texttt{Val\_Var}(v,k,l) & & t ::= \epsilon \\ \quad \mid \texttt{While\_unroll}(l,k) \mid \texttt{While\_iter}(l) \mid \texttt{Fun\_Call}(f,l) & & \quad \mid t.p \end{array}$$

where f is a function name, l a program point, v a program variable, k an integer.

For instance, the pre-token $\texttt{Fun\_Call}(f,l)$ characterizes traces that called the function f at point l and have not returned yet. The pre-token $\texttt{If\_true}(l)$ characterizes the traces that flowed through the true branch of the conditional at point l and have not reached the corresponding merge point yet. The pre-token $\texttt{Val\_Var}(v,k,l)$ characterizes the traces that have reached l with the condition $v = k$ satisfied and have not reached the corresponding merge point yet. The pre-token $\texttt{While\_unroll}(l,k)$ characterizes the traces that spent exactly k iterations in the loop at point l; the pre-token $\texttt{While\_iter}(l)$ characterizes the traces that spent more iterations in the loop than the number of unrolls for this loop.

A partition in the sense of Sect. 2.3 is defined as a tuple (l,t) where l is a control state and t a token, since we partition the system with the control states. In other words, we fix $E = \mathcal{L} \times \mathcal{T}$.

3.2 Abstract Values

Let us consider a control point l and a set P of partitions observed at this point, during the analysis. Then, the prefix of the tokens corresponding to the partitions in P can be shared. By construction, the set of tokens corresponding to the partition at a given program point is prefix-closed; hence, a local invariant is represented by a tree; a path from the root to a leaf in such a tree corresponds

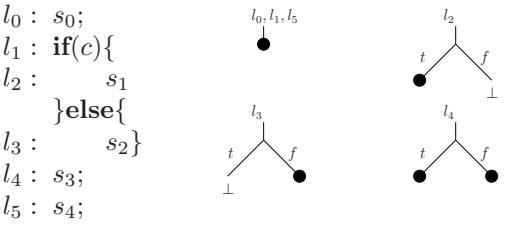
to a uniquely defined token. We write $D^\sharp$ for the underlying domain abstracting sets of states. We assume that $D^\sharp$ features a least element $\bot$ and a lub $\sqcup$.

Definition 6 (abstract value). *The* abstract partitioning domain *is defined as* $\mathbb{D}^\sharp = \mathcal{L} \to D_p^\sharp$*, with the pointwise ordering, where elements of* $D_p^\sharp$ *are defined by induction as follows:*

$$d \ ::= \ \mathrm{leaf}[v]\,(v \in D^\sharp) \ \mid \ \mathrm{node}[\phi]\,(\phi \in \mathcal{P} \to D_p^\sharp)$$

Roughly speaking, an element of $D_p^\sharp$ encodes a function $\mathcal{T} \to D^\sharp$; so an element of $\mathbb{D}^\sharp$ corresponds to an element of $\mathcal{L} \to (\mathcal{T} \to D^\sharp)$ (which is equivalent to $(\mathcal{L} \times \mathcal{T}) \to D^\sharp$). Hence, $\mathbb{D}^\sharp$ is obtained from the trace discriminating reachability domain (Sect. 2.3) by composition the abstraction $\gamma : D^\sharp \to \mathcal{P}(\mathcal{S})$ pointwise. It is a particular instance of a partitioning abstract domain as defined in Sect. 2.6. Furthermore, we insure sharing of common parts of the underlying numerical invariants across partitions, which reduces memory requirements.

Let us consider the program below (example of Sect 2.4 continued). We assume that the conditional at point l_1 is partitioned and that the corresponding merge-point is l_4. The shape of the partitions (hence, the local abstract invariants) at various program points is displayed below (the $\bullet$ symbol stands for a value in $D^\sharp$).



3.3 Iteration Scheme

The computation of an approximation of $[\![P]\!]$ requires some counterpart for the transition relation $\rightarrow$. For all $l, l' \in \mathcal{L}$, we let $\vartheta_{l,l'} : D_p^\sharp \to D_p^\sharp$ denote the abstract transfer function corresponding to the edge (l, l'). It is *sound* if and only if $\forall d \in D_p^\sharp,\ \forall \rho, \rho' \in \mathcal{M},\ \rho \in \gamma_p(d) \wedge (l, \rho) \rightarrow (l', \rho') \Rightarrow \rho' \in \gamma_p(\vartheta_{l,l'}(d))$.

In case $(\vartheta_{l,l'})$ is a family of sound abstract transfer functions, an abstract semantic function can be derived, such that $F \circ \gamma \subseteq \gamma \circ F^\sharp$; iterating it from the least element and using standard widening techniques yield a sound approximation of $[\![P]\!]_t$, i.e. the definition of a sound abstract semantics $[\![P]\!]^\sharp \in \mathbb{D}^\sharp$.

3.4 Abstract Transfer Functions

We let $\mathbb{C}$ denote the set of conditional expressions. We define the main abstract transfer functions below:

– **Non partitioning-related transfer functions**. we consider the case of the operator $\mathbf{guard} : \mathbb{C} \times D_p^\sharp \to D_p^\sharp$, which is the abstract counterpart of the concrete condition testing ($\mathbf{guard}_n$ is the operator provided by $D^\sharp$):

$$\begin{aligned} \mathbf{guard}(C, \text{leaf}[v]) &= \text{leaf}[\mathbf{guard}_n(C, v)] \\ \mathbf{guard}(D, \text{node}[\phi]) &= \text{node}[p \mapsto \mathbf{guard}(D, \phi(p))] \end{aligned}$$

– **Partition creation (create** $: (\mathcal{T} \to \mathbb{C}) \times D_p^\sharp$**):** if $C : \mathcal{T} \to \mathbb{C}$ is a family of conditions associated to all the created partitions, then $\mathbf{create}(C, d)$ creates a new partition defined by the condition $C(t)$ for each token t (Sect. 2.4). It can be written with an auxiliary function to accumulate prefixes:

$$\begin{aligned} \mathbf{create}(C, d) &= \mathbf{create}_0(C, \epsilon, d) \\ \mathbf{create}_0(C, t, \text{leaf}[v]) &= \text{node}[p \mapsto \text{leaf}[\mathbf{guard}_n(C(t), v)]] \\ \mathbf{create}_0(C, t, \text{node}[\phi]) &= \text{node}[p \mapsto \mathbf{create}_0(C, t.p, \phi(p))] \end{aligned}$$

– **Partition merge (merge** $: \mathcal{P}(\mathcal{T}) \times D_p^\sharp \to D_p^\sharp$**):** $\mathbf{merge}(X, d)$ yields a new abstract value whose partitions are elements of the set X (where the elements of X denote covering of the traces at the current program point and form another prefix-closed set of tokens); basically **merge** merges some existing partitions so as to restrict to a smaller set of partitions (Sect. 2.5). In practice X is a subset of the set of prefixes of the tokens corresponding to the partitions in d. It is defined in a similar way as **merge**:

$$\begin{aligned} \mathbf{merge}(X, d) &= \mathbf{merge}_0(X, \epsilon, d) \\ \mathbf{merge}_0(X, t, \text{leaf}[v]) &= \text{leaf}[v] \\ \mathbf{merge}_0(X, t, \text{node}[\phi]) &= \text{leaf}[\sqcup_n \{v \mid \text{node}[\phi] \text{ ancestor of leaf}[v]\}] \text{ if } t \in X \\ \mathbf{merge}_0(X, t, \text{node}[\phi]) &= \text{node}[p \mapsto \mathbf{merge}_0(X, t.p, \phi(p))] \quad \text{otherwise} \end{aligned}$$

The program displayed in Sect. 3.2 exemplifies partition creation (between l_1 and l_2, l_3) and partition merge (between l_4 and l_5).

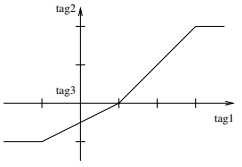
4 Trace Partitioning in Practice

The theoretical framework and our implementation of the trace partitioning abstract domain allow the introduction of a huge number of different partitions to refine the analysis. One last issue is to find which partition will indeed help the analysis while not impending too much on the complexity.

4.1 Manual Directives and Automatic Strategies

Our implementation allows the end-user to make such choices by specifying partitioning directives such as control flow or value partitions, or partition merges in the program code. Some functions to be partitioned according to the control flow (as in [17]) can also be specified (a merge is inserted at the return point). Although this possibility proved very useful for static analysis experts to improve the precision of an analysis, it is quite likely that most end-users would

l_0 : int $i = 0$;
l_1 : **while**($i < n$ && $x > tx[i+1]$) $\quad tc = \{0; 0.5; 1; 0\}$
l_2 : $\quad i++;$ $\quad tx = \{0; -1; 1; 3\}$
l_3 : $y = tc[i] \times (x - tx[i]) + ty[i]$ $\quad ty = \{-1; -0.5; -1; 2\}$
l_4 : ...



$$y = \begin{cases} -1 & \text{if } x \leq -1 \\ -0.5 + 0.5 \times x & \text{if } -1 \leq x \leq 1 \\ -1 + x & \text{if } 1 \leq x \leq 3 \\ 2 & \text{if } 3 \leq x \end{cases}$$

Fig. 2. Linear interpolation

miss opportunities to partition or propose too costly partitions. That is why we believe that static analyzer designers should also devise automatic strategies adapted to the kind of programs they wish to analyze precisely.

Automatic strategies for trace partitioning stem from imprecisions observed in the analysis of the target programs. When those imprecisions are understood, they can be captured by semantic patterns. In the three following sections, we present typical examples of imprecisions solved by a partitioning, together with the strategy which provides the ad-hoc partitions.

4.2 Linear Interpolations

Example: Computing functions defined by linear interpolation is a rather common task in real-time embedded software (e.g. functions describing the command reaction laws). We consider here the case of the piece of code below that inputs a value in variable x and computes in variable y the result of a linear interpolation. The loop checks in which range the variable x lies; then, the corresponding value is computed accordingly.

Non Relational Analysis: The execution of this fragment of code is expected to yield a value in the range $[-1, 2]$ whatever the value of x. However, inferring this most precise range is not feasible with a standard interval analysis, even if we partition the traces depending on the values of i at point l_3. Let us try with $-100 \leq x \leq 0$: then, we get $i \in \{0, 1\}$ at point l_3. The range for y at point l_4 is $[-0.5 + 0.5 \times (-100.), -0.5] \equiv [-50.5, -0.5]$ (this range is obtained in the case $i = 1$; the case $i = 0$ yields $y = -1$). Accumulating such huge imprecisions during the analysis may cause the properties of interest (e.g. the absence of runtime errors or the range of output values) not to be proved. We clearly see that some relations between the value of x and the value of i are required here.

Analysis with Trace Partitioning: Our approach to this case is to partition the traces according to the number of iterations in the loop. Indeed, if the loop is not iterated, then $i = 0$ at point l_3 and $x < -1$; if it is iterated exactly once,

then $i = 1$ at point l_3 and $-1 \leq x \leq 1$ and so forth. This approach yields the most precise range. Let us resume the analysis, with the initial constraint $-100 \leq x \leq 0$. The loop is iterated at most once and the partitions at point l_3 give:

- no iteration: $i = 0$; $x < -1$; $y = -1$
- one iteration: $i = 1$; $-1 \leq x \leq 0$; $-1 \leq y \leq -0.5$.

Therefore, the resulting range is $y \in [-1, -0.5]$, which is the optimal range (i.e. exactly the range of all values that could be observed in concrete executions).

The partitions generated in this example correspond to l_0, $(l_1, 0)$, $(l_2, 0)$, $(l_1, 1)$, $(l_2, 1)$, $(l_3, 0)$, $(l_3, 1)$, l_4; the partition associated to l_i is the set of traces ending at l_i; the partition associated with (l_i, j) is the set of traces ending at l_i after completing j iteration in the loop. This set of partitions is determined during the analysis, with directives requesting partitioning at point l_1 and merge at point l_4.

As we noted before, the trace partitioning turns out to be a reasonable alternative to the design of a more involved relational domain.

Strategy Implemented in Astrée**:** The imprecision observed when analyzing the linear interpolation have two causes: first, the expression at point l_3 computes the sum of two expressions which are not independent. Non-relational domains are quite imprecise in such cases (e.g. if $x \in [-1, 1]$, a dumb interval analysis will find $x - x \in [-2, 2]$). The second cause is that, through the use of arrays, the expression makes an implicit disjunction. Most efficient relational domains are imprecise on disjunctions (unions is usually the abstract operation that loses the most precision).

In Astrée, we use the following strategy to build partitions solving this kind of problem: first, we identify expressions e with an array access in a sum, such that another element of the sum is related to the index of the array access (it is the case for the expression at l_3, Fig 2). Then, we look backward for the last non-trivial assignment of that index. If it is in a loop, then we partition the loop, otherwise, we partition the values of the index after its assignment. In addition, we partition all the control flow between the index assignment and the expression e. We keep the analysis efficient by merging those partitions right after the expression e is used.

4.3 Barycenter

Finding precise invariants when analyzing divisions sharing a variable in the dividend and divider require either complex ad-hoc transfer functions (as in [12]) or guessing an appropriate linear form [3]. If the variable found in the dividend and divider ranges in a small set (less than, say, a thousand) we can get very precise results by partitioning the traces according to the dynamic values of that variable. Such partition will be quite cheap because its scope will be very local: it is only necessary to partition right before the assignment, and then we can merge right after the assignment.

A simple example using division is the computing a barycenter between two values. One expects the barycenter to be between those two values. But it is in fact a difficult task with classical abstract domains. In the following figure, we show an example of classical barycenter computation. As it is the case in many real-time embedded systems, this computation is inside an infinite loop.

l_0 : int $r = 0$; float $x = 0.0$;
l_1 : **while**(true){
l_2 : $r = \text{random}(0, 50)$;
l_3 : $x = (x * r + \text{random}(-100, 100))/(r + 1)$;
l_4 : }

Using non-relational domains, one cannot prove that x will never overflow, whereas it is a simple matter, partitioning the values during just one instruction, to prove that x stays in $[-100, 100]$. If we suppose $x \in [-100, 100]$ and $r \in [0, 50]$, we get $(x * r + \text{random}(-100, 100))/(r + 1)$ in $[-5100, 5100]$. whereas if we take any particular r in $[0, 50]$, we can compute that the expression is in $[-100, 100]$.

4.4 Loop Unrolling

Analyzing separately the n first iterations of a loop may greatly improve the precision of the final result, as is the case of the following examples:

- Some families of embedded programs –as those addressed by ASTRÉE– consist in a large loop; the initialization of some variables might be done during the first iteration(s), so it might be helpful to distinguish the first iterations during the analysis so as to take into account their particular role.
- Unrolling a loop might turn weak updates into strong updates. For instance, let us consider the program **for**$(i = 0; i < n; i = i + 1)\{t[i] = i\}$, which initializes an array t (we assume that all cells are equal to 0 before the loop). If we perform no unrolling of the loop, a simple interval analysis infers the interval constraint $i \in [0, n - 1]$; so the assignment to $t[i]$ is a weak update and in the end we get the family of constraints $\forall j,\ t[j] \in [0, n - 1]$.
  The body of the loop is very small; hence, the complete unrolling of all the iterations of the loop is not too costly. It leads to the much more precise family of constraints $\forall j,\ t[j] = j$.

In practice, defining the control point corresponding to the loop as the partitioning point and the control points right after the loop as the merging point leads to the unrolling of the n first iterations. This allows for more precise results in the analysis of the loop; yet does not make the analysis of the subsequent statements more costly.

The analysis of an unrolled loop starts with the computation of an invariant for the n first iterations; after that an invariant for all the following iterations is achieved thanks to the standard widening techniques. It is also possible to start with a partition of the whole loop, and decide during the computation of the invariants, that because of the growth of n, this partition might not be finite (or be too large) and thus, as described in Sect. 2.6, to use a coarser partition.

4.5 Experimental Results

We tested Astrée on a family of industrial size embedded codes. All partitions where chosen automatically. In the following table, we show the results for the analysis without partitioning and then **with partitioning**. For each program, we provide the size of the code as a number of LOCs, the number of iterations required for the analysis of the main loop (these programs all consist in a series of tasks executed at every clock tick, which is equivalent to a main loop), the analysis time in minutes (on a 3 GHz Bi-opteron, with 8 Gb of RAM), the memory consumption in megabytes and the number of alarms.

Program	test 1		test 2		test 3		test 4	
Code size (LOCs)	70 000		65 000		215 000		380 000	
Iterations	48	**43**	33	**32**	80	**59**	163	**62**
Analysis time (minutes)	44	**70**	21	**28**	180	**330**	970	**680**
Memory peak (Mb)	520	**550**	340	**390**	1 100	**1 300**	1 800	**2 200**
Alarms	658	**0**	552	**2**	4 963	**1**	6 693	**0**

The results show the expected positive impact on the precision, as the number of alarms of the analyzer is always reduced with partitioning; in all cases the analysis with partitioning results in a very low number of alarms whereas the analysis without partitioning yields huge numbers of false positives –much beyond what the end-user could check by hand. The analysis being more precise, less iterations to reach a post fixpoint are required with trace partitioning. In the case of test 4, the number of iterations required by the analysis with partitioning disabled even causes a much higher analysis time. Of course, using partitioning each iteration takes longer, but the cost in time and memory is very reasonable.

5 Conclusion

The partitioning of abstract domains was first introduced in [6]; it describes trace partitioning on the concrete level (sets of traces). We proposed to use such partitions to guide a restricted kind of disjunctions. Disjunctive completion usually gives very precise results, but has an exponential cost, that is why in practice, one must restrict the number of disjunctions. The idea of using the control flow to chose which disjunctions to keep was first introduced in [17], but still their proposal was not practical, especially for large programs. What we proposed here is a more general and flexible framework which allowed the Astrée static analyzer to be very precise on industrial programs [3].

Future work includes the extension of the partitioning abstract domain with backwards transfer functions, so as to do backwards analysis. A second extension would be to partition traces with the number of times a property of the memory state $\mathcal{P}$ was satisfied at a control point l, generalizing the condition-guided partitioning we presented here. This would allow expressing some kind of temporal properties of traces, by distinguishing traces that satisfied $\mathcal{P}$ at least once and the others.

Acknowledgments

We would like to thank the other members of the ASTRÉE team, Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné and David Monniaux for their help during the early developments of this work.

References

[1] AMMONS, G., AND LARUS, J. R. Improving data-flow analysis with path profiles. In *Conference on Programming Language Design and Implementation (PLDI'98)* (1998), ACM Press, pp. 72–84.

[2] BLANCHET, B., COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In *The Essence of Computation: Complexity, Analysis, Transformation*, T. Mogensen, D. Schmidt, and I. Sudborough, Eds., no. 2566 in LNCS. Springer-Verlag, 2002.

[3] BLANCHET, B., COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. A static analyzer for large safety-critical software. In *Conference on Programming Language Design and Implementation (PLDI'03)* (2003), ACM Press, pp. 196–207.

[4] BOURDONCLE, F. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming 4*, 2 (1992), 407–435.

[5] COUSOT, P. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes.* PhD thesis, Université de Grenoble, 1978.

[6] COUSOT, P. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981, ch. 10.

[7] COUSOT, P., AND COUSOT, R. Abstract intrepretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL'77)* (1977), ACM Press, pp. 238–252.

[8] COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *6th ACM Symposium on Principles of Programming Languages (POPL'79)* (1979), ACM Press, pp. 269–283.

[9] COUSOT, P., AND COUSOT, R. Basic concepts of abstract interpretation. In *Building the Information Society*. Kluwer Academic Publishers, 2004, ch. 4.

[10] COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. The astrée analyzer. In *European Symposium on Programming (ESOP'05)* (2005), This volume of LNCS, Springer-Verlag.

[11] COUSOT, P., AND HALBWACHS, N. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages (POPL'78)* (1978), ACM Press, pp. 84–97.

[12] FERET, J. Static analysis of digital filters. In *European Symposium on Programming (ESOP'04)* (2004), no. 2986 in LNCS, Springer-Verlag.

[13] GRANGER, P. Static Analysis of Arithmetical Congruences. *Int. J. Computer. Math. 30* (1989).

[14] Holley, L. H., and Rosen, B. K. Qualified data flow problems. In *7th ACM Symposium on Principles of Programming Languages (POPL'80)* (1980), ACM Press, pp. 69–82.
[15] Mauborgne, L. ASTRÉE: Verification of absence of run-time error. In *Building the Information Society*. Kluwer Academic Publishers, 2004, ch. 4.
[16] Miné, A. The octagon abstract domain. In *AST* (2001), IEEE, IEEE CS Press.
[17] Tzolowski, S., and Handjieva, M. Refining static analyses by trace-based partitionning using control flow. In *Static Analysis Symposium (SAS'98)* (1998), vol. 1503 of *LNCS*, Springer-Verlag.
[18] Venet, A. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *Static Analysis Symposium (SAS'96)* (1996), vol. 1145 of *LNCS*, Springer-Verlag.

The ASTRÉE Analyzer*

Patrick Cousot [2], Radhia Cousot [1,3], Jerôme Feret [2], Laurent Mauborgne [2], Antoine Miné [2], David Monniaux [1,2], and Xavier Rival [2]

[1] CNRS
[2] École Normale Supérieure, Paris, France
Firstname.Lastname@ens.fr
[3] École Polytechnique, Palaiseau, France
Firstname.Lastname@polytechnique.fr
http://www.astree.ens.fr/

Abstract. ASTRÉE is an abstract interpretation-based static program analyzer aiming at proving automatically the absence of run time errors in programs written in the C programming language. It has been applied with success to large embedded control-command safety critical real-time software generated automatically from synchronous specifications, producing a correctness proof for complex software without any false alarm in a few hours of computation.

1 Introduction

Software development, testing, use, and evolution is nowadays a major concern in many machine-driven human activities. Despite progress in the science of computing and the engineering of software aiming at developing larger and more complex systems, incorrect software is not so uncommon and sometimes quite problematic. Hence, the design of sound and efficient formal program verifiers, which has been a long-standing problem, is a grand challenge for the forthcoming decades.

All automatic proof methods involve some form of approximation of program execution, as formalized by abstract interpretation. They are sound but incomplete whence subject to *false alarms*, that is desired properties that cannot be proved to hold, hence must be signaled as potential problems, even though they do hold at runtime.

Although ASTRÉE addresses only part of the challenge, that of proving the absence of runtime errors in large embedded control-command safety critical real-time software generated automatically from synchronous specifications [1, 2, 3], it is a promising first step, in that it was able to make the correctness proof for large and complex software by abstract-interpretation based static analysis [4, 5] in a few hours of computations, without any false alarm.

* This work was supported in part by the French exploratory project ASTRÉE of the Réseau National de recherche et d'innovation en Technologies Logicielles (RNTL).

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 21–30, 2005.

2 Domain of Application of ASTRÉE

Synchronous C Programs. ASTRÉE can analyze C programs with pointers (including to functions), structures and arrays, integer and floating point computations, tests, loops, function calls, and branching (limited to forward `goto`, `switch`, `break`, `continue`). It excludes `union` types, dynamic memory allocation, unbounded recursive function calls, backward branching, conflicting side effects and the use of C libraries. This corresponds to a clean memory model and semantics as recommended for safety critical embedded real-time synchronous software for non-linear control of very complex control/command systems.

Semantics. The *concrete operational semantics* for the considered subset is that of the international C norm (ISO/IEC 9899:1999) instanced by implementation-specific behaviors depending upon the machine and compiler (e.g., representation and size of integers, IEEE 754-1985 norm for floats and doubles), restricted by user-defined programming guidelines (e.g., whether static variables can or cannot be assumed to be initialized to 0) and finally restricted by program-specific user requirements (such as static assertions). Programs may have a volatile environment where inputs are assumed to be safe (e.g., volatile floats cannot be `NaN`) and may be specified by a trusted configuration file (e.g., specifying physical restrictions on captor values or the maximum number of clock ticks, i.e., of calls to a `wait_for_clock()` function specific to synchronous systems). The *collecting semantics* is the set of partial traces for the concrete operational semantics starting from initial states. The *abstract semantics* is an abstraction of a trace-based refinement of the reachable states.

Specification. The absence of runtime errors is the implicit specification that there is no violation of the C norm (e.g., array index out of bounds), no implementation-specific undefined behaviors (e.g., floating-point division by zero), no violation of the programming guidelines (e.g., arithmetics operators on `short` variables should not overflow the range $[-32768, 32767]$ although, on the specific platform, the result can be well-defined through modular arithmetics), and no violation of the programmer-supplied assertions (which must all be statically verified). It follows that the only possible interrupts are clock ticks, an essential requirement of synchronous programs.

3 Characteristics of ASTRÉE

ASTRÉE is a *program analyzer* (it analyzes directly the program source and not some external specification or program model) which is *static* (the verification is performed before execution), entirely *automatic* (no end-user intervention is needed after parameterization by specialists for adaptation to a category of programs), *semantic-based* (unlike syntactic feature detectors in the spirit of `lint`), *sound* (it covers the whole state space and, contrarily to mere debuggers or bounded-trace software verifiers, never omits a potential error), *terminating*

(there is no possibility of non-termination of the analysis), and, in practice, has shown to be *efficient* (a few hours of computations for hundreds of thousands lines of code).

ASTRÉE is *multi-abstraction* in that it does not use a canonical abstraction but instead uses an approximate reduced cardinal product [5] of many numerical and symbolic abstract domains. The analyses performed by each abstract domain closely interact to perform mutual reductions. The abstraction is *specializable* in that new abstract domains can be easily included or useless ones excluded to adapt the analysis to a given category of programs. The design of ASTRÉE in `Ocaml` is *modular*. An instance of ASTRÉE is built by selecting `Ocaml` modules from a collection, each implementing an abstract domain. Most abstract domains are *infinitary* and *infinite-height*. We use widening/narrowing to enforce convergence. ASTRÉE is *specialized* to a safe programming style but is also *domain-aware* in that it knows about control/command (e.g., digital filters). Each abstract domain is *parametric* so that the precision/cost ratio can be tailored to user needs by options and/or directives in the code. The *automatic parameterization* enables the generation of parametric directives in the code to be programmed. ASTRÉE can therefore be specialized to perform fully automatically for each specific application domain. This design structure makes ASTRÉE both fast and very *precise*: there are very few or no false alarms when conveniently adapted to an application domain. It follows that ASTRÉE is a *formal verifier* that scales up.

4 Design of ASTRÉE by Refinement

ASTRÉE was designed starting from a simple memory model (with references to abstract variables representing either a single or multiple concrete memory locations) and an interval abstraction ($a \leq X \leq b$ where X is a variable and a, b are constants to be determined by the analysis), which is precise enough to express the absence of runtime errors. The widening uses thresholds [1]. This is extremely fast (if sufficient care has been taken to use good data structures) but quite imprecise. Then, numerous abstract domains were designed and experimented until an acceptable cost/precision ratio was obtained. Sometimes, a more precise domain results in an improvement in both analysis precision *and* time (most often because the number of iterations is reduced).

5 The ASTRÉE Fixpoint Iterator

The fixpoint computation of the invariant post-fixpoint [4] is by structural induction on the program abstract syntax, keeping a minimal number of abstract invariants. Functions are handled by semantic expansion of the body (thus excluding unbounded recursion) while convergence is accelerated by non-monotonic widening/narrowing for loops [4, 2]. It follows that the abstract fixpoint transformer is non-monotonic, which is not an issue as it abstracts monotonic concrete fixpoints [6]. Because abstract domains are themselves implemented using

floats, possible rounding errors may produce instabilities in the post-fixpoint check which can be solved thanks to perturbations [2, 7, 8]. Finally, the specification checking is performed by a forward propagation of the stable abstract post-fixpoint invariant.

6 Examples of Numerical Abstractions in ASTRÉE

The Domain of Octagons. An example of numerical abstract domain is the weakly relational domain of octagons [9, 10, 7] ($\pm X \pm Y \leq a$ where X, Y are variables and a is a constant to be determined by the analysis).

```
volatile int vD, vX;                __ASTREE_volatile_input((vD [0,16]));
void main () {                      __ASTREE_volatile_input((vX [-128,128]));
 int D, X, Y = 0, R, S;
 while (1) {
  X = vX; D = vD;
  S = Y; R = X - S; Y = X;
  if (R <= -D) { Y = S - D; }
  else if (D <= R) { Y = S + D; }
 }}
```

Fig. 1. Rate limiter and configuration file

For instance [7], at each loop iteration of the rate limiter of Fig. 1, a new value for the entry X is fetched within $[-128, 128]$ and a new maximum rate D is chosen in $[0, 16]$. The program then computes an output Y that tries to follow X but is compelled to change slowly: the difference between Y and its value in the preceding iteration is bounded, in absolute value, by the current value of D. The state variable S is used to remember the value of Y at the last iteration while R is a temporary variable used to avoid computing the difference $X - S$ twice. A relational domain is necessary to prove that the output Y is bounded by the range $[-128, 128]$ of X, which requires the discovery of the invariant $R = X - S$. The octagon abstract domain will discover a weaker property, $R + S \in [-128, 128]$, which is precise enough to prove that $Y \in [-M, M]$ is stable whenever $M \geq 144$. So, by widening, M will be set to the least threshold greater than 144 which is loose but precise enough to prove the absence of runtime errors (indeed ASTRÉE finds $Y \in [-261, 261]$). This example is out of the scope of the interval domain.

Heterogeneous Structural Abstraction. The use of the domain of octagons in ASTRÉE is an example of heterogeneous abstraction which depends upon the program structure and is not the same at each program point. Indeed, the octagonal abstraction would be too costly to handle, e.g., thousands of global variables at each program point. The domain of octagons is therefore parameterized by packs of variables attached to blocks/functions by way of directives. These packs specify which variables should be candidate for octagonal analysis in the given block/function. The determination of accurate packs would require

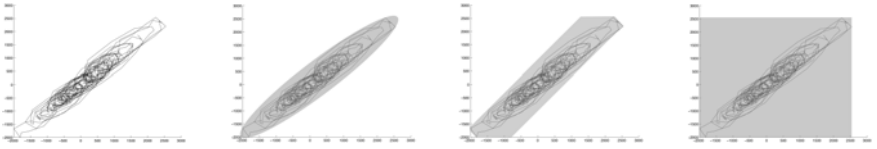


Fig. 2. Filter trace Ellipsoid abstraction Octagon abstraction Interval abstraction

a huge amount of work, if done by hand. Therefore the packing parameterization is automatized using context-sensitive syntactic criteria. Experimentations show that the average pack size is usually of order of 3 or 4 variables while the number of packs grows linearly with the program size. It follows that precise abstractions are performed only when needed, which is necessary to scale up.

Floating-Point Interval Linear Form Abstraction. A general problem with relational numerical domains is that of floating point numbers. Considering them as reals (as usually done with theorem provers) or fixed point numbers (as in CBMC [11]) would not conform to the norm whence would be unsound. Using rationals or other symbolic reals in the abstract domains would be too costly. The general approach [7, 8] has been to define the concrete semantics of floating point computations in the reals (taking the worst possible rounding errors explicitly into account), to abstract with real numbers but to implement, thanks to a further sound over-approximation, using floats. For example the float expression $(x + y) + z$ is evaluated as in the reals as $x + y + z + \varepsilon_1 + \varepsilon_2$ where $|\varepsilon_1| \leq \epsilon_{\text{rel}}.|x + y| + \epsilon_{\text{abs}}$ and $|\varepsilon_2| \leq \epsilon_{\text{rel}}.|x + y + \varepsilon_1 + z| + \epsilon_{\text{abs}}$. The real ε_1 encodes rounding errors in the atomic computation $(x + y)$, and the real ε_2 encodes rounding errors in the atomic computation $(x + y + \varepsilon_1) + z$. The parameters ϵ_{rel} and ϵ_{abs} depends on the floating-point type being used in the analyzed program. This *linearization* [7, 8] of arbitrary expressions is a correct abstraction of the floating point semantics into interval linear forms $[a_0, b_0] + \sum_{k=1}^{n} [a_k, b_k] X_k$. This approach separates the treatment of rounding errors from that of the numerical abstract domains.

The Simplified Filter Abstract Domains. The simplified filter abstract domains [13] provide examples of domain-aware abstractions. A typical example of simplified filter behavior is traced in Fig. 2 (tracing the sequence `D1` in Fig. 3). Interval and octagonal envelops are unstable because they are rotated and shrunk a little at each iteration so that some corner always sticks out of the envelop. However, the ellipsoid of Fig. 2 is stable. First, filter domains use dynamical linear properties that are captured by the other domains such as the range of input variables (`x1` and `y1` for the example of Fig. 3) and symbolic affine equalities with interval coefficients (to model rounding errors) such as $\texttt{t1} \in [1 - \varepsilon_1, 1 + \varepsilon_1].\texttt{x1} + [b1[0] - \varepsilon_2, b1[0] + \varepsilon_2].\texttt{D1[0]} - [b1[1] - \varepsilon_3, b1[1] + \varepsilon_3].\texttt{D1[1]} + [-\varepsilon, \varepsilon]$ for the example of Fig. 3 (where ε_1, ε_2, and ε_3 describe relative error contributions and ε describes an absolute error contribution). These symbolic equalities are captured either by *linearization* (see Sect. 6), or by symbolic constant propagation (see Sect. 7). Then, simplified filter domains infer non linear properties and compute bounds on the range of output variables (`t1` and `t2` in Fig. 2). For the example

```
float A1[3] = { 1, 0.5179422053046, 1.0 };
float b1[2] = { 1.470767736573, 0.5522073405779 };
float A2[3] = { 1, 1.633101801841, 1.0 };
float b2[2] = { 1.742319554830, 0.820939679242 };
float D1[2], D2[2]; float P, X; volatile float E;
void iir4(float *x, float *y)
{ float x1, y1, t1, t2;
  x1 = 0.0117749388721091* *x; t1 = x1 + b1[0]*D1[0] - b1[1]*D1[1];
  y1 = A1[0]*t1 - A1[1]*D1[0] + A1[2]*D1[1]; D1[1] = D1[0]; D1[0] = t1;
  t2 = y1 + b2[0]*D2[0] - b2[1]*D2[1];
  *y = A2[0]*t2 - A2[1]*D2[0] + A2[2]*D2[1]; D2[1] = D2[0];  D2[0] = t2;
  __ASTREE_log_vars((P,y1,x1,t2;ellipse))
}
int main () { while (1) { X = E; iir4(&X,&P); __ASTREE_log_vars((P)); }}
```

Fig. 3. Fourth order Infinite Impulse Response (IIR) filter [12]

of Fig. 3, ASTRÉE over-approximates the interval of variation of `D2[0]` by $[-6890.23, 6890.23]$, which is precise enough to prove the absence of overflow.

On the Limits of User-Provided Assertions. The filter ellipsoidal abstraction illustrates the limits of user provided assertions. Even if the user injects the correct bounds, as an interval information, for all filter outputs, the interval domain cannot exploit them as they are not stable. To reach zero false alarm, the abstract domains should be able to express a loop invariant which is strong enough to be inductive and to imply the absence of runtime errors. User assertions are therefore useful only when they refer to assertions expressible in the abstract domains of a static analyzer. They are mainly useful to provide widening/narrowing limits but techniques such as widenings with thresholds are even more convenient.

On the Limits of Automatic Refinement. The filter ellipsoidal abstraction shows the limits of automatic refinement strategies based on counter-examples. From a finite series of counter-examples to the stability of intervals or octagons, the refinement procedure would have to automatically discover the ellipsoidal abstraction and infer the corresponding sophisticated data representations and algorithms.

The Arithmetic-Geometric Progression Abstract Domain. In synchronous programs, the arithmetic-geometric progression abstract domain [14] can be used to estimate ranges of floating point computations that may diverge in the long run due to rounding errors (although they may be stable in the reals) thanks to a relation with the clock, which, for physical systems that cannot run for ever, must be bounded. In the example of Fig. 4, the bound of `B` is:

$$|\mathtt{B}| \leq a * ((20. + b/(a-1)) * (a)^{\mathrm{clock}} - b/(a-1)) + b \leq 30.7191369175$$

where $a = 1.00000011921$ and $b = 5.87747175411e-39$.

```
volatile float E,T;                    __ASTREE_volatile_input((T[-1.0,1.0]));
float A,B,X;                           __ASTREE_volatile_input((E[-20.0,20.0]));
int main () {                          __ASTREE_max_clock((3600000));
 while (1) {
  if (T>0){X = E;}
  else {X = B;}
  B = B-(((2.0*B)-A-X)*0.005);
  A = X;
  __ASTREE_wait_for_clock(());
 }}
```

Fig. 4. Geometric progression and configuration file

Relative Precision of Abstract Domains. The arithmetic-geometric progression abstract domain provides an example of sophisticated domain that can advantageously replace a simpler domain, specifically the *clock domain* [1], formerly used in ASTRÉE, relating each variable X to the bounded clock C (incremented on clock ticks) as intervals for $X - C$ and $X + C$ to indirectly bound X from the user-provided bound on the clock C.

7 Examples of Symbolic Abstractions in ASTRÉE

Memory Abstraction. A first example of symbolic domain in ASTRÉE is the memory abstraction model shortly described in [2].

The Symbolic Constant Domain. The symbolic constant domain [7, 8] is reminiscent of Kildall's constant propagation abstract domain, that is the smash product of infinitely many domains of the form $\bot \sqsubset e \sqsubset \top$, but memorizes symbolic expressions e instead of numerical constants. It keeps track of symbolic relations between variables and performs simplifications (such as simplifying `Z=X; Y=X-Z` into `Z=X; Y=0`, which does appear in mechanically generated programs). Such relational information is essential for the interval abstract domain (e.g., to derive that $Y = 0$). Again the abstract domain is parameterized (e.g., by simplification strategies).

The Boolean Relation Domain. The Boolean relation domain [2] copes with the use of booleans in the control of synchronous programs. It is a reduced cardinal power [5] with boolean base implemented as a decision tree with sharing (à la BDD) and exponents at the leaves. It is parametric in the maximum number of boolean variables and in the packs of variables which are involved at the leaves. The maximum number 3 was determined experimentally and the packing is automatized.

The Expanded Filter Abstract Domains. The expanded filter abstract domains associate recursive sequence definitions to tuples of numerical variables automatically detected by the analysis [13]. For instance, a second order

filter is encoded by a recursive definition of the form $S_{n+2} = a.S_{n+1} + b.S_n + c.E_{n+2} + d.E_{n+1} + e.E_n$ ((E_n) denotes an input stream and (S_n) denotes an output stream). The second order filter domain relates abstract values M to the quadruples of variables (V, W, X, Y) detected by the analysis. This symbolic property means that there exists a positive integer p and a recursive sequence satisfying $S_{n+2} = a.S_{n+1} + b.S_n + c.E_{n+2} + d.E_{n+1} + e.E_n$, for any positive integer n, such that:

- $V = S_{p+1}$, $W = S_p$, $X = E_{p+1}$, and $Y = E_p$;
- the abstract value M gives an abstraction of the values S_0 and S_1 and of the sequence (E_n).

We compute a bound on the value of V, by unfolding the recursive definition several times (so that we describe the contribution of the last inputs very accurately). The contributions of rounding errors and of previous inputs are bounded by using a simplified filter domain (the ellipsoid domain [13] in our example).

Trace Partitioning. Trace partitioning [15] is a local parametric symbolic abstraction of sets of traces, which is a local refinement of reachable states. By relative completeness of Floyd's proof method, it is useless to reason on traces and sets of states should be precise enough. However, this greatly simplifies the abstraction which would otherwise require to establish more relations among variables. Examples are loop unrolling, case analysis for tests, etc.

8 Performances of ASTRÉE

ASTRÉE has been applied with success to large embedded control-command safety critical real-time software generated automatically from synchronous specifications, producing a correctness proof for complex software without any false alarm in a few hours of computations (see Fig. 5).

Nb of lines	70 000	226 000	400 000
Number of iterations	32	51	88
Memory	599 Mb	1.3 Gb	2.2 Gb
Time	46mn	3h57mn	11h48mn
False alarms	0	0	0

Fig. 5. Performance of ASTRÉE (64 bits monoprocessor)

9 ASTRÉE Visualisator

According to the desired information, it is possible to serialize the invariants and alarms attached by ASTRÉE to program points, blocks, loops or functions

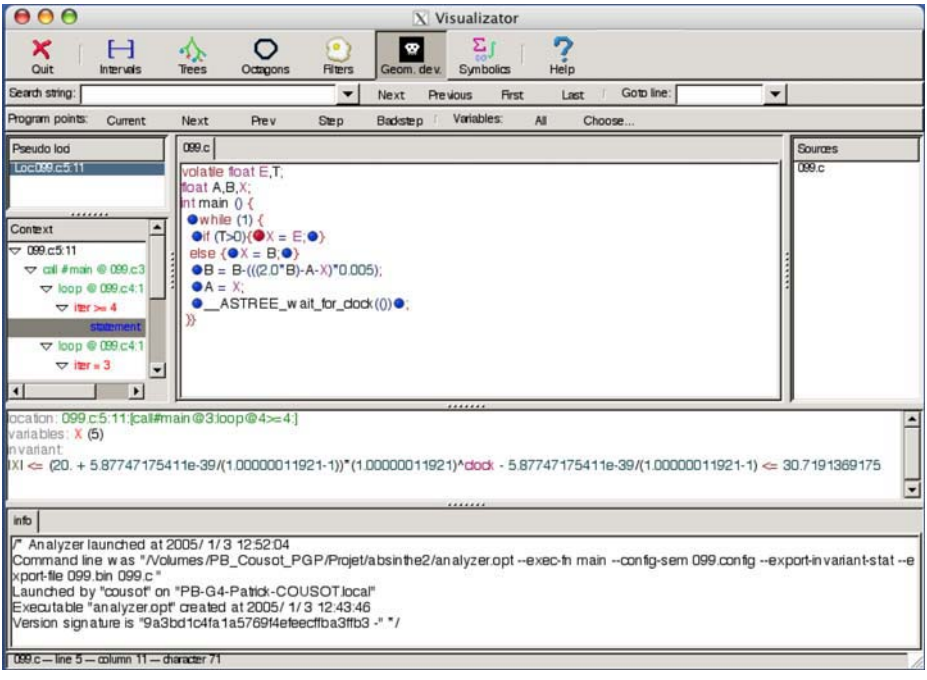


Fig. 6. Vizualisator

and to vizualize them, per abstract domain, using a graphical vizualisator to navigate in the program invariants as shown on Fig. 6.

10 Conclusion and Future Work

Abstract interpretation-based static analyzers have recently shown to scale-up for different programming languages and different areas of application [16, 17]. ASTRÉE is certainly the first static analyzer able to *fully prove automatically* the absence of runtime errors in real-life large industrial synchronous programs. It is therefore a verifier (as opposed to a debugger or testing aid). In case of erroneous source programs, an assistance to error source localization is presently being incorporated in ASTRÉE thanks to backward analyses. By extension of the abstract domains, ASTRÉE can be extended beyond synchronous programs.

To go beyond and generalize to more complex memory models and asynchronous programs will necessitate a complete redesign of the basic memory abstraction and fixpoint iterators. This will be the object of ASTRÉE successors.

Acknowledgements. We warmly thank Bruno Blanchet for his contribution to ASTRÉE.

References

1. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In Mogensen, T., Schmidt, D., Sudborough, I., eds.: The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. LNCS 2566. Springer (2002) 85–108
2. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proc. ACM SIGPLAN '2003 Conf. PLDI, San Diego, ACM Press (2003) 196–207
3. Mauborgne, L.: ASTRÉE: Verification of absence of run-time error. In Jacquart, P., ed.: Building the Information Society. Kluwer Academic Publishers (2004) 385–392
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4^{th} ACM POPL. (1977) 238–252
5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: 6^{th} ACM POPL. (1979) 269–282
6. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. of Logic and Comput. **2** (1992) 511–547
7. Miné, A.: Weakly Relational Numerical Abstract Domains. Thèse de doctorat en informatique, École polytechnique, Palaiseau, France (2004)
8. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In Schmidt, D., ed.: Proc. 30^{th} ESOP '2004, Barcelona. LNCS 2986, Springer (2004) 3–17
9. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In Danvy, O., Filinski, A., eds.: Proc. 2^{nd} Symp. PADO '2001. Århus, 21–23 May 2001, LNCS 2053, Springer (2001) 155–172
10. Miné, A.: The octagon abstract domain. In: AST 2001 in WCRE 2001. IEEE, IEEE CS Press (2001) 310–319
11. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In Jensen, K., Podelski, A., eds.: Proc.10th Int. Conf. TACAS '2004, Barcelona. LNCS 2988, Springer (2004) 168–176
12. Aamodt, T., Chow, P.: Numerical error minimizing floating-point to fixed-point ANSI C compilation. In: 1st Workshop on Media Processors and DSPs. (1999) 3–12
13. Feret, J.: Static analysis of digital filters. In Schmidt, D., ed.: Proc. 30^{th} ESOP '2004, Barcelona. LNCS 2986, Springer (2004) 33–48
14. Feret, J.: The arithmetic-geometric progression abstract domain. In Cousot, R., ed.: Proc. 6^{th} VMCAI '2005, Paris. LNCS 3385, Springer (2004) 2–58
15. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In Sagiv, M., ed.: Proc. 31^{th} ESOP '2005, Edinbugh. This volume of LNCS., Springer (2005)
16. Alt, M., C., F., Martin, F., Wilhelm, R.: Cache behavior prediction by abstract interpretation. In Cousot, R., Schmidt, D., eds.: Proc. 3^{rd} Int. Symp. SAS '96. Aachen, 24–26 sept. 1996, LNCS 1145. Springer (1996) 52–66
17. Venet, A., Brat, G.: Precise and efficient static array bound checking for large embedded C programs. In: Proc. ACM SIGPLAN '2004 Conf. PLDI, Washington DC, ACM Press (2004) 231–242

Interprocedural Herbrand Equalities

Markus Müller-Olm[1], Helmut Seidl[2], and Bernhard Steffen[1]

[1] Universität Dortmund, FB 4, LS V, 44221 Dortmund, Germany
{mmo, steffen}@ls5.cs.uni-dortmund.de
[2] TU München, Lehrstuhl für Informatik II, 80333 München, Germany
seidl@in.tum.de

Abstract. We present an aggressive interprocedural analysis for inferring value equalities which are independent of the concrete interpretation of the operator symbols. These equalities, called *Herbrand equalities*, are therefore an ideal basis for truly machine-independent optimizations as they hold on every machine. Besides a general *correctness* theorem, covering arbitrary call-by-value parameters and local and global variables, we also obtain two new *completeness* results: one by constraining the analysis problem to *Herbrand constants*, and one by allowing *side-effect-free functions* only. Thus if we miss a constant/equality in these two scenarios, then there exists a separating interpretation of the operator symbols.

1 Introduction

Analyses for finding definite equalities between variables or variables and expressions in a program have been used in program optimization for a long time. Knowledge about definite equalities can be exploited for performing and enhancing powerful optimizing program transformations. Examples include constant propagation, common subexpression elimination, and branch elimination [3, 8], partial redundancy elimination and loop-invariant code motion [18, 22, 12], and strength reduction [23]. Clearly, it is undecidable whether two variables always have the same value at a program point even without interpreting conditionals [17]. Therefore, analyses are bound to detect only a subset, i.e., a safe approximation, of all equivalences. Analyses based on the *Herbrand interpretation* of operator symbols consider two values equal only if they are constructed by the same operator applications. Such analyses are said to detect *Herbrand equalities*. Herbrand equalities are precisely those equalities which hold independent of the interpretation of operators. Therefore, they are an ideal basis for truly machine-independent optimizations as they hold on every machine, under all size restrictions, and independent of the chosen evaluation strategy.

In this paper, we propose an aggressive interprocedural analysis of Herbrand equalities. Note that a straight-forward generalization of intraprocedural inference algorithms to programs with procedures using techniques along the lines of [7, 20, 13] fails since the domain of Herbrand equalities is obviously infinite. Besides a general *correctness* theorem, covering arbitrary call-by-value parameters and local and global variables, we also obtain two new *completeness* results: One by constraining the analysis problem to *Herbrand constants*, and one by allowing *side-effect-free functions* only. Thus if we miss

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 31–45, 2005.

a constant/equality in these constrained scenarios, then a separating interpretation of the operator symbols can be constructed.

For reasons of exposition, we treat the case of side-effect-free functions, which constitutes an interesting class of programs in its own, separately first. The key technical idea here is to abstract the effect of a function call $\mathbf{x}_1 := f(\mathbf{x}_1, \ldots, \mathbf{x}_k)$, $\mathbf{x}_i$ program variables, by a *conditional* assignment, i.e., a pair $(\phi, \mathbf{x}_1 := e)$ consisting of a precondition ϕ together with an assignment $\mathbf{x}_1 := e$, e some term, where ϕ is a conjunction of Herbrand equalities. If the precondition is satisfied, the function call behaves like the assignment $\mathbf{x}_1 := e$, otherwise, like an assignment of an unknown value. The interesting observation is that for functions without side-effects, this is not only *sound*, i.e., infers only valid Herbrand equalities between variables, but also *complete*, i.e., infers for every program point u *all* equalities which are valid at u. In fact, our algorithm is the first inter-procedural analysis of Herbrand equalities which is complete on this class of programs. Moreover, its running time asymptotically coincides with that of the best intraprocedural algorithms for the same problem [22, 9]. Technically, the conditional assignments for functions are determined by effective weakest precondition computations for particular postconditions. For side-effect-free functions, the postcondition takes the form $\mathbf{y} \doteq \mathbf{x}_1$ where $\mathbf{y}$ is a fresh variable and $\mathbf{x}_1$ is the variable that receives the return value of the function. In the next step, we generalize this analysis to functions with *multiple return values*. Such functions correspond to procedures accessing and modifying multiple global variables. The resulting analysis is sound; moreover, we prove that it is strong enough to find all *Herbrand constants*, i.e., determines for every program point u all equalities $\mathbf{x}_j \doteq t$ for variables $\mathbf{x}_j$ and ground terms t.

Related Work. Early work on detecting equalities without considering the meaning of the operator symbols dates back to Cocke and Schwartz [4]. Their technique, the famous *value numbering*, was developed for basic blocks and assigns hash values to computations. While value numbering can be rather straightforwardly extended to forking programs, program joins pose nontrivial problems, because the concept of value equality based on equal hash numbers is too fine granular. In his seminal paper [11], Kildall presents a generalization that extends Cocke's and Schwartz's technique to flow graphs with loops by explicitly representing the equality information on terms in form of *partitions*, which allows one to treat joins of basic blocks in terms of intersection. This gave rise to a number of algorithms focusing on efficiency improvement [17, 1, 3, 19, 8, 10].

The connection of the originally pragmatic techniques to the Herbrand interpretation has been established in [21] and Steffen et al. [22], which present provably *Herbrand complete* variants of Kildall's technique and a compact representation of the Herbrand equalities in terms of *structured partition DAGs (SPDAGs)*. Even though these DAGs provide a redundancy-free representation, they still grow exponentially in the number of program terms. This problem was recently attacked by Gulwani and Necula, who arrived at a polynomial algorithm by showing that SPDAGs can be pruned, if only equalities of bounded size are of interest [9]. This observation can also be exploited for our structurally rather different interprocedural extension.

Let us finally mention that all this work abstracts conditional branching by nondeterministic choice. In fact, if equality guards are taken into account then determining whether a specific equality holds at a program point becomes undecidable [15]. Dis-

equality constraints, however, can be dealt with intraprocedurally [15]. Whether or not inter-procedural extensions are possible is still open.

The current paper is organized as follows. In Section 2 we introduce un-interpreted programs with side-effect-free functions as the abstract model of programs for which our Herbrand analysis is complete. In Section 3 we collect basic facts about conjunctions of Herbrand equalities. In Section 4 we present the weakest precondition computation to determine the effects of function calls. In Section 5 we use this description of effects to extend an inference algorithm for intraprocedurally inferring all valid Herbrand equalities to deal with side-effect-free functions as well. In Section 6 we generalize the approach to a sound analysis for procedures accessing global variables and indicate that it infers all Herbrand constants. Finally, in Section 7 we summarize and describe further directions of research.

2 Herbrand Programs

We model programs by systems of nondeterministic flow graphs that can recursively call each other as in Figure 1. Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ be the set of variables the program operates on. We assume that the basic statements in the program are either assignments of the form $\mathbf{x}_j := t$ for some expression t possibly involving variables from $\mathbf{X}$ or *nondeterministic* assignments $\mathbf{x}_j := ?$ and that branching in general is nondeterministic. Assignments $\mathbf{x}_j := \mathbf{x}_j$ have no effect onto the program state. They can be used as skip statements as, e.g., at the right edge from program point 4 to 5 in Figure 1 and also to abstract guards. Nondeterministic assignments $\mathbf{x}_j := ?$ safely abstract statements in a source program our analysis cannot handle, for example input statements.

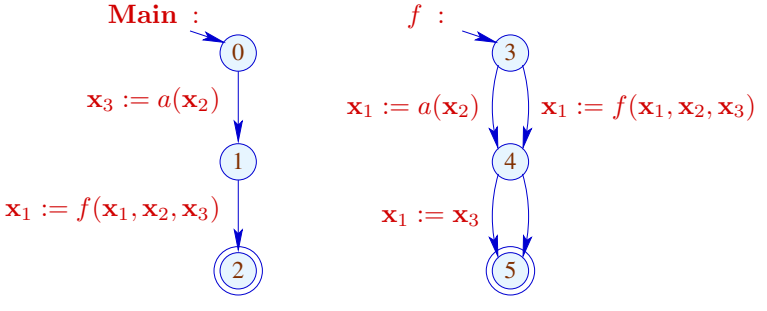


Fig. 1. A small Herbrand program

A *program* comprises a finite set Funct of *function names* that contains a distinguished function **Main**. First, we consider side-effect-free functions with call-by-value parameters and single return values. Without loss of generality, every call to a function f is of the form: $\mathbf{x}_1 := f(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ — meaning that the values of all variables are passed to f as actual parameters, and that the variable $\mathbf{x}_1$ always receives the return

value of f which is the final value of $\mathbf{x}_1$ after execution of f.[1] In the body of f, the variables $\mathbf{x}_2, \ldots, \mathbf{x}_k$ serve as local variables. More refined calling conventions, e.g., by using designated argument variables or passing the values of expressions into formal parameters can easily be reduced to our case. Due to our standard layout of calls, each call is uniquely represented by the name f of the called function. In Section 6, we will extend our approach to procedures which read and modify *global* variables. These globals will be the variables $\mathbf{x}_1, \ldots, \mathbf{x}_m$, $m \leq k$. Procedures f are then considered as functions computing *vector assignments* $(\mathbf{x}_1, \ldots, \mathbf{x}_m) := f(\mathbf{x}_1, \ldots, \mathbf{x}_k)$.

Let Stmt be the set of assignments and calls. Program execution starts with a call to $\mathbf{Main}$. Each function name $f \in \mathsf{Funct}$ is associated with a *control flow graph* $G_f = (N_f, E_f, \mathsf{st}_f, \mathsf{ret}_f)$ that consists of a set N_f of *program points*; a set of edges $E_f \subseteq N_f \times \mathsf{Stmt} \times N_f$; a special *entry (or start) point* $\mathsf{st}_f \in N_f$; and a special *return point* $\mathsf{ret}_f \in N_f$. We assume that the program points of different functions are disjoint: $N_f \cap N_g = \emptyset$ for $f \neq g$. This can always be enforced by renaming program points. Moreover, we denote the set of edges labeled with assignments by Base and the set of edges calling some function f by Call.

We consider *Herbrand interpretation* of terms, i.e., we maintain the structure of expressions but abstract from the concrete meaning of operators. Let Ω denote a signature consisting of a set Ω_0 of constant symbols and sets $\Omega_r, r > 0$, of operator symbols of rank r which possibly may occur in right-hand sides of assignment statements or values. Let $\mathcal{T}_\Omega$ the set of all formal terms built up from Ω. For simplicity, we assume that the set Ω_0 is non-empty, and there is at least one operator. Note that under this assumption, the set $\mathcal{T}_\Omega$ is infinite. Let $\mathcal{T}_\Omega(\mathbf{X})$ denote the set of all terms with constants and operators from Ω which additionally may contain occurrences of variables from $\mathbf{X}$. Since we do not interpret constants and operators, a *state* assigning values to the variables is conveniently modeled by a mapping $\sigma : \mathbf{X} \to \mathcal{T}_\Omega$. Such mappings are also called *ground substitutions*. Accordingly, the effect of one execution of a function can be represented by a term $e \in \mathcal{T}_\Omega(\mathbf{X})$ which describes how the result value for variable $\mathbf{x}_1$ is constructed from the values of the variables $\mathbf{x}_1, \ldots, \mathbf{x}_k$ before the call. Note that such effects nicely can be accumulated from the rear where every assignment $\mathbf{x}_j := t$ extends the effect by substituting t for variable $\mathbf{x}_j$.

We define the collecting semantics of a program which will be abstracted in the sequel. Every assignment $\mathbf{x}_j := t$ induces a transformation $[\![\mathbf{x}_j := t]\!] : 2^{\mathbf{X} \to \mathcal{T}_\Omega} \to 2^{\mathbf{X} \to \mathcal{T}_\Omega}$ of the set of program states *before* the assignment into the set of states after the assignment, and a transformation $[\![\mathbf{x}_j := t]\!] : 2^{\mathcal{T}_\Omega(\mathbf{X})} \to 2^{\mathcal{T}_\Omega(\mathbf{X})}$ of the set of function effects accumulated *after* the assignment into the effects including the assignment:

$$[\![\mathbf{x}_j := t]\!]\, S = \{\sigma[\mathbf{x}_j \mapsto \sigma(t)] \mid \sigma \in S\} \qquad [\![\mathbf{x}_j := t]\!]\, T = \{e[t/\mathbf{x}_j] \mid e \in T\}$$

Here $\sigma(t)$ is the term obtained from t by replacing each occurrence of a variable $\mathbf{x}_i$ by $\sigma(\mathbf{x}_i)$ and $\sigma[\mathbf{x}_j \mapsto t']$ is the substitution that maps $\mathbf{x}_j$ to $t' \in \mathcal{T}_\Omega$ and $\mathbf{x}_i \neq \mathbf{x}_j$ to $\sigma(\mathbf{x}_i)$. Moreover, $e[t/\mathbf{x}_j]$ denotes the result of substituting t in e for variable $\mathbf{x}_j$. Similarly, we have two interpretations of the non-deterministic assignment $\mathbf{x}_j := ?$:

$$[\![\mathbf{x}_j := ?]\!]\, S = \bigcup\{[\![\mathbf{x}_j := c]\!]\, S \mid c \in \mathcal{T}_\Omega\} = \{\sigma[\mathbf{x}_j \mapsto \sigma(c)] \mid c \in \mathcal{T}_\Omega, \sigma \in S\}$$
$$[\![\mathbf{x}_j := ?]\!]\, T = \bigcup\{[\![\mathbf{x}_j := c]\!]\, T \mid c \in \mathcal{T}_\Omega\} = \{e[c/\mathbf{x}_j] \mid c \in \mathcal{T}_\Omega, e \in T\}$$

[1] Alternatively, we could view the variable $\mathbf{x}_1$ as one global variable which serves as scratch pad for passing information from a called procedure back to its caller.

Thus, $\mathbf{x}_j := ?$ is interpreted as the non-deterministic choice between *all* assignments of values to $\mathbf{x}_j$. In a similar way, we reduce the semantics of calls to the semantics of assignments, here to the variable $\mathbf{x}_1$. For determining the sets of reaching states, we introduce a binary operator $[\![\mathsf{call}]\!] : 2^{\mathcal{T}_\Omega(\mathbf{X})} \times 2^{\mathbf{X}\to\mathcal{T}_\Omega} \to 2^{\mathbf{X}\to\mathcal{T}_\Omega}$ which uses a set of effects of the called function to transform the set of states before the call into the set of states after the call. For transforming sets of effects, we rely on a binary operator $[\![\mathsf{call}]\!] : 2^{\mathcal{T}_\Omega(\mathbf{X})} \times 2^{\mathcal{T}_\Omega(\mathbf{X})} \to 2^{\mathcal{T}_\Omega(\mathbf{X})}$ which takes the effects of a called function to extend the effects accumulated after the call. We define:

$$\begin{array}{ll} [\![\mathsf{call}]\!]\,(T,S) & = \bigcup\{[\![\mathbf{x}_1 := t]\!]\,S \mid t \in T\} \quad = \{\sigma[\mathbf{x}_1 \mapsto \sigma(t)] \mid t \in T, \sigma \in S\} \\ [\![\mathsf{call}]\!]\,(T_1,T_2) & = \bigcup\{[\![\mathbf{x}_1 := t]\!]\,T_2 \mid t \in T_1\} = \{e[t/\mathbf{x}_1] \mid t \in T_1, e \in T_2\} \end{array}$$

Thus, a call is interpreted as the non-deterministic choice between all assignments $\mathbf{x}_1 := t$ where t is a potential effect of the called function. We use the operators $[\![\ldots]\!]$ to characterize the sets of effects of functions, $\mathbf{S}(f) \subseteq \mathcal{T}_\Omega(\mathbf{X})$, $f \in \mathsf{Funct}$, by means of a constraint system $\mathbf{S}$:

$$\begin{array}{lll} [\mathrm{S1}] & \mathbf{S}(f) & \supseteq \mathbf{S}(\mathsf{st}_f) \\ [\mathrm{S2}] & \mathbf{S}(\mathsf{ret}_f) & \supseteq \{\mathbf{x}_1\} \\ [\mathrm{S3}] & \mathbf{S}(u) & \supseteq [\![s]\!]\,(\mathbf{S}(v)) \qquad \text{if } (u,s,v) \in \mathsf{Base} \\ [\mathrm{S4}] & \mathbf{S}(u) & \supseteq [\![\mathsf{call}]\!]\,(\mathbf{S}(f),\mathbf{S}(v)) \quad \text{if } (u,f,v) \in \mathsf{Call} \end{array}$$

Note that the effects are accumulated in sets $\mathbf{S}(u) \subseteq \mathcal{T}_\Omega(\mathbf{X})$ for program points u from the *rear*, i.e., starting from the return points. Calls are dealt with by constraint [S4]. If the ingoing edge (u,f,v) is a call to a function f, we extend the terms already found for v with the potential effects of the called function f by means of the operator $[\![\mathsf{call}]\!]$. Obviously, the operators $[\![\mathbf{x}_j := t]\!]$ and hence also the operators $[\![\mathbf{x}_j := ?]\!]$ and $[\![\mathsf{call}]\!]$ are monotonic (even distributive). Therefore, by Knaster-Tarski's fixpoint fixpoint theorem, the constraint system $\mathbf{S}$ has a unique least solution whose components (for simplicity) are denoted by $\mathbf{S}(u), \mathbf{S}(f)$ as well.

We use the effects $\mathbf{S}(f)$ of functions and the operators $[\![...]\!]$ to characterize the sets of *reaching program states*, $\mathbf{R}(u), \mathbf{R}(f) \subseteq (\mathbf{X} \to \mathcal{T}_\Omega)$, by a constraint system $\mathbf{R}$:

$$\begin{array}{lll} [\mathrm{R1}] & \mathbf{R}(\mathbf{Main}) & \supseteq \mathbf{X} \to \mathcal{T}_\Omega \\ [\mathrm{R2}] & \mathbf{R}(f) & \supseteq \mathbf{R}(u)\,, \qquad \text{if } (u,f,\_) \in \mathsf{Call} \\ [\mathrm{R3}] & \mathbf{R}(\mathsf{st}_f) & \supseteq \mathbf{R}(f) \\ [\mathrm{R4}] & \mathbf{R}(v) & \supseteq [\![s]\!]\,(\mathbf{R}(u))\,, \qquad \text{if } (u,s,v) \in \mathsf{Base} \\ [\mathrm{R5}] & \mathbf{R}(v) & \supseteq [\![\mathsf{call}]\!]\,(\mathbf{S}(f),\mathbf{R}(u))\,, \quad \text{if } (u,f,v) \in \mathsf{Call} \end{array}$$

Again, since all occurring operators are monotonic (even distributive), this constraint system has a unique least solution whose components are denoted by $\mathbf{R}(u)$ and $\mathbf{R}(f)$.

3 Herbrand Equalities

A substitution $\sigma : \mathbf{X} \to \mathcal{T}_\Omega(\mathbf{X})$ (possibly containing variables in the image terms) satisfies a conjunction of equalities $\phi \equiv s_1 \doteq t_1 \wedge \ldots \wedge s_m \doteq t_m$ (where $s_i, t_i \in \mathcal{T}_\Omega(\mathbf{X})$ and "$\doteq$" a formal equality symbol) iff $\sigma(s_i) = \sigma(t_i)$ for $i = 1, \ldots, m$. Then we also write $\sigma \models \phi$. We say, ϕ is valid at a program point u iff it is valid for all states $\sigma \in \mathbf{R}(u)$.

As we rely on Herbrand interpretation here, an equality which is valid at a program point u is also called a valid *Herbrand equality* at u.

Let us briefly recall some basic facts about conjunctions of equations. A conjunction ϕ is *satisfiable* iff $\sigma \models \phi$ for at least one σ. Otherwise, i.e., if ϕ is unsatisfiable, ϕ is logically equivalent to false. This value serves as the bottom value of the lattice we use in our analysis. The greatest value is given by the *empty* conjunction which is always true and therefore also denoted by true. The ordering is by logical implication "$\Rightarrow$". Whenever the conjunction ϕ is satisfiable, then there is a *most general* satisfying substitution σ, i.e., $\sigma \models \phi$ and for every other substitution τ satisfying ϕ, $\tau = \tau_1 \circ \sigma$ for some substitution τ_1. Such a substitution is often also called a *most general unifier* of ϕ. In particular, this means that the conjunction ϕ is equivalent to $\bigwedge_{\mathbf{x}_i \neq \sigma(\mathbf{x}_i)} \mathbf{x}_i \doteq \sigma(\mathbf{x}_i)$. Thus, every satisfiable conjunction of equations is equivalent to a (possibly empty) finite conjunction of equations $\mathbf{x}_{j_i} \doteq t_i$ where the left-hand sides $\mathbf{x}_{j_i}$ are distinct variables and none of the equations is of the form $\mathbf{x}_j \doteq \mathbf{x}_j$. Let us call such conjunctions *reduced*. The following fact is crucial for proving termination of our proposed fixpoint algorithms.

Proposition 1. *For every sequence $\phi_0 \Leftarrow \ldots \Leftarrow \phi_m$ of pairwise inequivalent conjunctions ϕ_j using k variables, $m \leq k + 1$.* □

Proposition 1 follows since for satisfiable reduced non-equivalent conjunctions ϕ_i, ϕ_{i+1}, $\phi_i \Leftarrow \phi_{i+1}$ implies that ϕ_{i+1} contains strictly more equations than ϕ_i.

In order to construct an abstract lattice of properties, we consider *equivalence classes* of conjunctions of equations which, however, will always be represented by one of their members. Let $\mathbb{E}(\mathbf{X}')$ denote the set of all (equivalence classes of) finite reduced conjunctions of equations with variables from $\mathbf{X}'$. This set is partially ordered w.r.t. "$\Rightarrow$" (on the representatives). The pairwise greatest lower bound always exists and is given by conjunction "$\wedge$". Since by Proposition 1, all descending chains in this lattice are ultimately stable, not only finite but also infinite subsets $X \subseteq \mathbb{E}(\mathbf{X}')$ have a greatest lower bound. Hence, $\mathbb{E}(\mathbf{X}')$ is a *complete* lattice.

4 Weakest Preconditions

For reasoning about return values of functions, we introduce a fresh variable $\mathbf{y}$ and determine for every function f the weakest precondition, $\mathbf{WP}(f)$, of the equation $\mathbf{y} \doteq \mathbf{x}_1$ w.r.t. f. Given that the set of effects of f equals $T \subseteq \mathcal{T}_\Omega(\mathbf{X})$, the weakest precondition of $\mathbf{y} \doteq \mathbf{x}_1$ is given by $\bigwedge\{\mathbf{y} \doteq e \mid e \in T\}$ – which is equivalent to a finite conjunction due to the compactness property of Proposition 1. Intuitively, true as precondition means that the function f has an empty set of effects only, whereas $\phi' \wedge \mathbf{y} \doteq e$ expresses that the single value returned for $\mathbf{x}_1$ is e — under the assumption that ϕ' holds. Thus, ϕ' implies all equalities $e \doteq e'$, $e' \in T$. In particular, if ϕ' is unsatisfiable, i.e., equivalent to false, then the function may return different values.

For computing preconditions, we will work with the subset $\mathbb{E}_{\mathbf{y}}$ of $\mathbb{E}(\mathbf{X} \cup \{\mathbf{y}\})$ of (equivalence classes of) conjunctions ϕ of equalities with variables from $\mathbf{X} \cup \{\mathbf{y}\}$ which are either equivalent to true or equivalent to a conjunction $\phi' \wedge \mathbf{y} \doteq e$ for some $e \in \mathcal{T}_\Omega(\mathbf{X})$. We can assume that ϕ' does not contain y, since any occurrence of $\mathbf{y}$ in ϕ' can be replaced with e. We introduce a function $\alpha_{\mathbf{S}} : 2^{\mathcal{T}_\Omega(\mathbf{X})} \to \mathbb{E}_{\mathbf{y}}$ by:

$$\alpha_{\mathbf{S}}(T) = \bigwedge_{e \in T} (\mathbf{y} \doteq e)$$

By transforming arbitrary unions into conjunctions, $\alpha_{\mathbf{S}}$ is an *abstraction* in the sense of [6]. Our goal is to define abstract operators $[\![\mathbf{x}_j := t]\!]^\sharp$, $[\![\mathbf{x}_j := ?]\!]^\sharp$ and $[\![\mathsf{call}]\!]^\sharp$.

A precondition $[\![\mathbf{x}_j := t]\!]^\sharp\, \phi$ of a conjunction of equalities ϕ for an assignment $\mathbf{x}_j := t$ can be obtained by the well-known rule:

$$[\![\mathbf{x}_j := t]\!]^\sharp \phi = \phi[t/\mathbf{x}_j]$$

where $\phi[t/\mathbf{x}_j]$ denotes the formula obtained from ϕ by substituting t for $\mathbf{x}_j$. This transformation returns the *weakest* precondition for the assignment. The transformer for non-deterministic assignments is reduced to the transformation of assignments:

$$[\![\mathbf{x}_j := ?]\!]^\sharp \phi = \bigwedge_{c \in \mathcal{T}_\Omega} [\![\mathbf{x}_j := c]\!]^\sharp \phi = \bigwedge_{c \in \mathcal{T}_\Omega} \phi[c/\mathbf{x}_j]$$

By assumption, $\mathcal{T}_\Omega$ contains at least two elements $t_1 \neq t_2$. If ϕ contains $\mathbf{x}_j$, then $\phi[t_1/\mathbf{x}_j] \wedge \phi[t_2/\mathbf{x}_j]$ implies $t_1 \doteq t_2$ (because we are working with Herbrand interpretation) which is false by the choice of t_1, t_2. Hence, the transformer can be simplified to:

$$[\![\mathbf{x}_j := ?]\!]^\sharp \phi \;=\; \phi[t_1/\mathbf{x}_j] \wedge \phi[t_2/\mathbf{x}_j] \;=\; \begin{cases} \mathsf{false} & \text{if } \mathbf{x}_j \text{ occurs in } \phi \\ \phi & \text{otherwise} \end{cases}$$

The first equation means that $\mathbf{x}_j := ?$ is semantically equivalent (w.r.t. weakest preconditions of Herbrand equalities) to the nondeterministic choice between the two assignments $\mathbf{x}_j := t_1$ and $\mathbf{x}_j := t_2$.

In order to obtain safe preconditions for calls, we introduce a binary operator $[\![\mathsf{call}]\!]^\sharp$. In the first argument, this operator takes a precondition ϕ_1 of a function body for the equation $\mathbf{y} \doteq \mathbf{x}_1$. The second argument of $[\![\mathsf{call}]\!]^\sharp$ is a postcondition ϕ_2 after the call. We define:

$$\begin{aligned} [\![\mathsf{call}]\!]^\sharp(\mathsf{true}, \phi_2) &= \mathsf{true} \\ [\![\mathsf{call}]\!]^\sharp(\phi' \wedge (\mathbf{y} \doteq e), \phi_2) &= \begin{cases} \phi' \wedge \phi_2[e/\mathbf{x}_1] & \text{if } \mathbf{x}_1 \text{ occurs in } \phi_2 \\ \phi_2 & \text{otherwise} \end{cases} \end{aligned}$$

If the weakest precondition of $\mathbf{y} \doteq \mathbf{x}_1$ is true, we return true, since a set of effects is abstracted with true only if it is empty. In order to catch the intuition of the second rule of the definition, first assume that ϕ' is true. This corresponds to the case where the abstracted set of effects consists of a single term e only. The function call then is semantically equivalent to the assignment $\mathbf{x}_1 := e$. Accordingly, our definition gives: $[\![\mathsf{call}]\!]^\sharp(\mathbf{y} \doteq e, \phi_2) = \phi_2[e/\mathbf{x}_1]$. In general, different execution paths may return different terms e' for $\mathbf{x}_1$. The precondition ϕ' then implies that all these e' equal e. If ϕ_2 does not contain $\mathbf{x}_1$, ϕ_2 is not affected by assignments to $\mathbf{x}_1$ anyway. Therefore in this case, $[\![\mathsf{call}]\!]^\sharp(\phi_1, \phi_2) = \phi_2$. If on the other hand, ϕ_2 contains the variable $\mathbf{x}_1$, then ϕ_2 holds after the call provided $\phi_2[e/\mathbf{x}_1]$ holds before the call as well as ϕ'.

The definition of $[\![\mathsf{call}]\!]^\sharp(\phi_1, \phi_2)$ is independent of the chosen representation of ϕ_1. To see this, assume that ϕ_1 is also equivalent to $\phi_1' \wedge (\mathbf{y} \doteq t_1)$ for some ϕ_1', t_1 not containing $\mathbf{y}$. Then in particular, $\phi' \wedge (\mathbf{y} \doteq t)$ implies $\mathbf{y} \doteq t_1$ as well as ϕ_1' from which we deduce that ϕ' also implies $t \doteq t_1$. Therefore, $\phi' \wedge \phi_2[t/\mathbf{x}_1]$ implies $\phi_1' \wedge \phi_2[t_1/\mathbf{x}_1]$. By exchanging the roles of ϕ', t and ϕ_1', t_1 we find the reverse implication and the equivalence follows. We establish the following distributivity properties:

Proposition 2. *1.* $[\![\mathbf{x}_j := t]\!]^\sharp$ *preserves* true *and distributes over "$\wedge$".*
2. $[\![\mathbf{x}_j := ?]\!]^\sharp$ *preserves* true *and distributes over "$\wedge$".*
3. In each argument, the operation $[\![\mathsf{call}]\!]^\sharp$ *preserves* true *and distributes over "$\wedge$".*

Proof: Assertion 1 holds since substitutions preserve true and commute with "$\wedge$". Assertion 2 follows from 1, since $[\![\mathbf{x}_j := ?]\!]^\sharp\,\phi = ([\![\mathbf{x}_j := t_1]\!]^\sharp\,\phi) \wedge ([\![\mathbf{x}_j := t_2]\!]^\sharp\,\phi)$ for ground terms $t_1 \neq t_2$. For the third assertion, the statement concerning the second argument of $[\![\mathsf{call}]\!]^\sharp$ is straightforwardly verified from the definition. The same is true for the preservation of true in the first argument. It remains to verify that

$$[\![\mathsf{call}]\!]^\sharp(\phi_1 \wedge \phi_2, \phi) = [\![\mathsf{call}]\!]^\sharp(\phi_1, \phi) \wedge [\![\mathsf{call}]\!]^\sharp(\phi_2, \phi)$$

If either ϕ_1 or ϕ_2 equal false, the assertion is obviously true. The same holds if either ϕ_1 or ϕ_2 equal true. Otherwise, we can assume that for $i = 1, 2$, ϕ_i is satisfiable, reduced and of the form: $\phi_i' \wedge (\mathbf{y} \doteq e_i)$ for some ϕ_i' not containing $\mathbf{y}$. If ϕ does not contain $\mathbf{x}_1$, the assertion is again trivially true. Therefore, we additionally may assume that ϕ contains at least one occurrence of $\mathbf{x}_1$. Then by definition, $[\![\mathsf{call}]\!]^\sharp(\phi_i, \phi) = \phi_i' \wedge \phi[e_i/\mathbf{x}_1]$, and we obtain:

$$\begin{aligned}[\![\mathsf{call}]\!]^\sharp(\phi_1, \phi) \wedge [\![\mathsf{call}]\!]^\sharp(\phi_2, \phi) &= \phi_1' \wedge \phi[e_1/\mathbf{x}_1] \wedge \phi_2' \wedge \phi[e_2/\mathbf{x}_1] \\ &= \phi_1' \wedge \phi_2' \wedge (e_1 \doteq e_2) \wedge \phi[e_1/\mathbf{x}_1]\end{aligned}$$

since ϕ contains an occurrence of $\mathbf{x}_1$. On the other hand, we may also rewrite $\phi_1 \wedge \phi_2$ to: $\phi_1' \wedge \phi_2' \wedge (e_1 \doteq e_2) \wedge (\mathbf{y} \doteq e_1)$ where only the last equation contains $\mathbf{y}$. Therefore:

$$[\![\mathsf{call}]\!]^\sharp(\phi_1 \wedge \phi_2, \phi) = \phi_1' \wedge \phi_2' \wedge (e_1 \doteq e_2) \wedge \phi[e_1/\mathbf{x}_1]$$

which completes the proof. □

We construct a constraint system $\mathbf{WP}$ for preconditions of functions by applying the abstraction function $\alpha_{\mathbf{S}}$ to the constraint system $\mathbf{S}$ for collecting effects of functions. Thus, we replace $\{\mathbf{x}_1\}$ with $(\mathbf{y} \doteq \mathbf{x}_1)$ and the operators $[\![\ldots]\!]$ with $[\![\ldots]\!]^\sharp$. We obtain:

$$\begin{array}{lll}
[\mathbf{WP}1] & \mathbf{WP}(f) & \Rightarrow \mathbf{WP}(\mathsf{st}_f) \\
[\mathbf{WP}2] & \mathbf{WP}(\mathsf{ret}_f) & \Rightarrow (\mathbf{y} \doteq \mathbf{x}_1) \\
[\mathbf{WP}3] & \mathbf{WP}(u) & \Rightarrow [\![s]\!]^\sharp(\mathbf{WP}(v))\,, \qquad \text{if } (u, s, v) \in \mathsf{Base} \\
[\mathbf{WP}4] & \mathbf{WP}(u) & \Rightarrow [\![\mathsf{call}]\!]^\sharp(\mathbf{WP}(f), \mathbf{WP}(v))\,, \text{ if } (u, f, v) \in \mathsf{Call}
\end{array}$$

By Knaster-Tarski fixpoint theorem, the constraint system $\mathbf{WP}$ has a greatest solution w.r.t. "$\Rightarrow$" which we denote with $\mathbf{WP}(f), \mathbf{WP}(u)$, $f \in \mathsf{Funct}, u \in N$. With Proposition 2, we verify that $\alpha_{\mathbf{S}}$ has the following properties:

1. $\alpha_{\mathbf{S}}(\{\mathbf{x}_1\}) = (\mathbf{y} \doteq \mathbf{x}_1)$;
2. $\alpha_{\mathbf{S}}([\![\mathbf{x}_j := t]\!]\,T) = [\![\mathbf{x}_j := t]\!]^\sharp(\alpha_{\mathbf{S}}(T))$;
3. $\alpha_{\mathbf{S}}([\![\mathbf{x}_j := ?]\!]\,T) = [\![\mathbf{x}_j := ?]\!]^\sharp(\alpha_{\mathbf{S}}(T))$;
4. $\alpha_{\mathbf{S}}([\![\mathsf{call}]\!](T_1, T_2)) = [\![\mathsf{call}]\!]^\sharp(\alpha_{\mathbf{S}}(T_1), \alpha_{\mathbf{S}}(T_2))$.

By the Transfer Lemma from fixpoint theory (c.f., e.g., [2, 5]), we therefore find:

Theorem 1 (Weakest Preconditions). *Let p be a program of size n with k variables.*

1. *For every function f of p, $\mathbf{WP}(f) = \bigwedge\{(\mathbf{y} \doteq t) \mid t \in \mathbf{S}(f)\}$; and for every program point u of p, $\mathbf{WP}(u) = \bigwedge\{(\mathbf{y} \doteq t) \mid t \in \mathbf{S}(u)\}$.*
2. *The greatest solution of constraint system $\mathbf{WP}$ can be computed in time $\mathcal{O}(n \cdot k \cdot \Delta)$ where Δ is the maximal size of a DAG representation of a conjunction occurring during the fixpoint computation.* □

Thus, the greatest solution of the constraint system $\mathbf{WP}$ precisely characterizes the weakest preconditions of the equality $\mathbf{x}_1 \doteq \mathbf{y}$. Evaluation of "$\wedge$" as well as of a right-hand side in the constraint system $\mathbf{WP}$ at most doubles the sizes of DAG representations of occurring conjunctions. Therefore, the value Δ is bounded by $2^{\mathcal{O}(n \cdot k)}$.

Example 1. Consider the function f from Figure 1. First, f and every program point is initialized with the top element true of the lattice $\mathbb{E}_{\mathbf{y}}$. The first approximation of the weakest precondition at program point 4 for $\mathbf{y} \doteq \mathbf{x}_1$ at 5, then is:

$$\mathbf{WP}(4) \;=\; (\mathbf{y} \doteq \mathbf{x}_1) \wedge (\llbracket \mathbf{x}_1 := \mathbf{x}_3 \rrbracket^\sharp\, (\mathbf{y} \doteq \mathbf{x}_1) \;=\; (\mathbf{y} \doteq \mathbf{x}_1) \wedge (\mathbf{y} \doteq \mathbf{x}_3)$$

Accordingly, we obtain for the start point 3,

$$\begin{aligned}\mathbf{WP}(3) &= \llbracket \mathsf{call} \rrbracket^\sharp(\mathsf{true}, \mathbf{WP}(4)) \wedge (\llbracket \mathbf{x}_1 := a(\mathbf{x}_2) \rrbracket^\sharp\, (\mathbf{WP}(4)))\\ &= \mathsf{true} \wedge (\mathbf{y} \doteq a(\mathbf{x}_2)) \wedge (\mathbf{y} \doteq \mathbf{x}_3)\\ &= (\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{y} \doteq \mathbf{x}_3)\end{aligned}$$

Thus, we obtain $(\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{y} \doteq \mathbf{x}_3)$ as a first approximation for the weakest precondition of $\mathbf{y} \doteq \mathbf{x}_1$ w.r.t. f. Since the fixpoint computation already stabilizes here, we have found that $\mathbf{WP}(f) \;=\; (\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{y} \doteq \mathbf{x}_3)$. □

5 Inferring Herbrand Equalities

For computing weakest preconditions, we have relied on conjunctions of equalities, (pre-) ordered by "$\Rightarrow$" where the greatest lower bound was implemented by the logical "$\wedge$". For *inferring* Herbrand equalities, we again use conjunctions of equalities, now over the set of variables $\mathbf{X}$ alone, i.e., we use $\mathbb{E} = \mathbb{E}(\mathbf{X})$ — but now we resort to least upper bounds (instead of greatest lower bounds). Conceptually, the least upper bound $\phi_1 \sqcup \phi_2$ of two elements in $\mathbb{E}$ corresponds to the best approximation of the disjunction $\phi_1 \vee \phi_2$. Thus, it is the conjunction of all equalities implied both by ϕ_1 and ϕ_2. We can restrict ourselves to equalities of the form $\mathbf{x}_i \doteq t$ ($\mathbf{x}_i \in \mathbf{X}, t \in \mathcal{T}_\Omega(\mathbf{X})$). Accordingly,

$$\begin{aligned}\phi_1 \sqcup \phi_2 &= \bigwedge\{\mathbf{x}_j \doteq t \mid (\phi_1 \vee \phi_2) \Rightarrow (\mathbf{x}_j \doteq t)\}\\ &= \bigwedge\{\mathbf{x}_j \doteq t \mid (\phi_1 \Rightarrow (\mathbf{x}_j \doteq t)) \wedge (\phi_2 \Rightarrow (\mathbf{x}_j \doteq t))\}\end{aligned}$$

Consider, e.g., $\phi_1 \equiv (\mathbf{x}_1 \doteq g(a(\mathbf{x}_3))) \wedge (\mathbf{x}_2 \doteq a(\mathbf{x}_3))$ and $\phi_2 \equiv (\mathbf{x}_1 \doteq g(b)) \wedge (\mathbf{x}_2 \doteq b)$. Then $\phi_1 \sqcup \phi_2$ is equivalent to $\mathbf{x}_1 \doteq g(\mathbf{x}_2)$.

Conjunctions of equalities are not closed under existential quantification. Therefore, we introduce the operators $\exists^\sharp \mathbf{x}_j$ as the best approximations to $\exists\, \mathbf{x}_j$ in $\mathbb{E}$:

$$\begin{aligned}\exists^\sharp \mathbf{x}_j.\phi &= \bigwedge\{\mathbf{x}_i \doteq t \mid i \neq j, (\exists\, \mathbf{x}_j.\phi) \Rightarrow (\mathbf{x}_i \doteq t), t \text{ does not contain } \mathbf{x}_j\}\\ &= \bigwedge\{\mathbf{x}_i \doteq t \mid i \neq j, \phi \Rightarrow (\mathbf{x}_i \doteq t), t \text{ does not contain } \mathbf{x}_j\}\end{aligned}$$

So, for example, $\exists^\sharp \mathbf{x}_2.\, (\mathbf{x}_1 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{x}_3 \doteq b(a(\mathbf{x}_2), c)) \;=\; \mathbf{x}_3 \doteq b(\mathbf{x}_1, c)$

We readily verify that "$\exists^\sharp \mathbf{x}_j$" preserves false and commutes with "$\sqcup$". The operations "$\sqcup$" and "$\exists^\sharp\, \mathbf{x}_j$" can be efficiently implemented on *partition DAG* representations [22] . More specifically, '$\exists^\sharp\, \mathbf{x}_j$" is linear-time whereas the least upper bound of two conjunctions with DAG representations of sizes n_1, n_2 can be performed in time $\mathcal{O}(n_1 + n_2)$ resulting in (a DAG representation of) a conjunction of size $\mathcal{O}(n_1 + n_2)$.

We define the abstraction $\alpha_{\mathbf{R}} : 2^{\mathbf{X} \to \mathcal{T}_\Omega} \to \mathbb{E}$ that maps a set of states to the conjunction of all equalities valid for all states in the set:

$$\alpha_{\mathbf{R}}(S) = \bigwedge\{x_j \doteq t \mid \forall \sigma \in S : \sigma \models x_j \doteq t\}$$

As an equality holds for a state $\sigma : \mathbf{X} \to \mathcal{T}_\Omega$ iff it is implied by the conjunction $\mathbf{x}_1 \doteq \sigma(\mathbf{x}_1) \wedge \ldots \wedge \mathbf{x}_k \doteq \sigma(\mathbf{x}_k)$ we have $\alpha_{\mathbf{R}}(S) = \bigsqcup\{\bigwedge_{i=1}^k \mathbf{x}_i \doteq \sigma(\mathbf{x}_i) \mid \sigma \in S\}$. In particular, this implies that $\alpha_{\mathbf{R}}$ commutes over unions.

We must provide abstractions of the operators $\llbracket \ldots \rrbracket$. We define:

$$\begin{aligned}
[\![\mathbf{x}_j := t]\!]^\sharp\phi &= \exists^\sharp\mathbf{y}.\phi[\mathbf{y}/\mathbf{x}_j] \wedge (\mathbf{x}_j \doteq t[\mathbf{y}/\mathbf{x}_j]) \\
[\![\mathbf{x}_j := ?]\!]^\sharp\phi &= \bigsqcup\{[\![\mathbf{x}_j := c]\!]^\sharp\phi \mid c \in \mathcal{T}_\Omega\} \\
&= \bigsqcup\{\exists^\sharp\mathbf{y}.\phi[\mathbf{y}/\mathbf{x}_j] \wedge (\mathbf{x}_j \doteq c]) \mid c \in \mathcal{T}_\Omega\} \\
&= \exists^\sharp\mathbf{y}.\bigsqcup\{\phi[\mathbf{y}/\mathbf{x}_j] \wedge (\mathbf{x}_j \doteq c]) \mid c \in \mathcal{T}_\Omega\} \\
&= \exists^\sharp\mathbf{y}.\phi[\mathbf{y}/\mathbf{x}_j] \\
&= \exists^\sharp\mathbf{x}_j.\phi
\end{aligned}$$

Thus, $[\![\mathbf{x}_j := t]\!]^\sharp\phi$ is the best abstraction of the strongest postcondition of ϕ w.r.t. $\mathbf{x}_j := t$ and the abstract transformer $[\![\mathbf{x}_j := ?]\!]^\sharp$ is given by abstract existential quantification. For instance, we have: $[\![\mathbf{x}_1 := \mathbf{x}_3]\!]^\sharp\,(\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \;=\; (\mathbf{x}_1 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{x}_3 \doteq a(\mathbf{x}_2))$ and $[\![\mathbf{x}_3 := ?]\!]^\sharp\,(\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{x}_1 \doteq a(\mathbf{x}_2)) \;=\; (\mathbf{x}_1 \doteq a(\mathbf{x}_2))$. These definitions provide obvious implementations using partition DAGs. In particular, the abstract transformer $[\![\mathbf{x}_j := t]\!]^\sharp$ can be computed in time linear in the size n_1 of the argument and the size n_2 of (a DAG representation of) t. Moreover, the DAG representation of the result is again of size $\mathcal{O}(n_1 + n_2)$. A similar estimation also holds for nondeterministic assignments.

The crucial point in constructing an analysis is the abstract operator $[\![\mathsf{call}]\!]^\sharp$ for function calls. The first argument of this operator takes the weakest precondition ϕ_1 of $(\mathbf{y} \doteq \mathbf{x}_1)$ for a (possibly empty) set of effects of some function. The second argument ϕ_2 takes a conjunction of equalities which is valid before the call. We define:

$$\begin{aligned}
[\![\mathsf{call}]\!]^\sharp(\mathsf{true}, \phi_2) &= \mathsf{false} \\
[\![\mathsf{call}]\!]^\sharp(\phi' \wedge (\mathbf{y} \doteq e), \phi_2) &= \begin{cases} [\![\mathbf{x}_1 := e]\!]^\sharp\phi_2 & \text{if} \quad \phi_2 \Rightarrow \phi' \\ [\![\mathbf{x}_1 := ?]\!]^\sharp\phi_2 & \text{otherwise} \end{cases}
\end{aligned}$$

The first rule states that everything is true at an unreachable program point. Otherwise, we can write ϕ_1 as $\phi' \wedge (\mathbf{y} \doteq e)$ where ϕ' and e do not contain $\mathbf{y}$. If ϕ' is implied by the precondition ϕ_2, we are guaranteed that all return values for $\mathbf{x}_1$ are equivalent to e. In this case, the call behaves like an assignment $\mathbf{x}_1 := e$. Otherwise, at least two different return values are possible. Then we treat the function call like a non-deterministic assignment $\mathbf{x}_1 := ?$.

Example 2. Consider, e.g., the call of function f in **Main** in Fig. 1. By Example 1, $\mathbf{WP}(f)$ equals $\phi_1 = (\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{y} \doteq \mathbf{x}_3)$. Before the call, $\phi_2 = (\mathbf{x}_3 \doteq a(\mathbf{x}_2))$ holds. Accordingly, we obtain:

$$\begin{aligned}
[\![\mathsf{call}]\!]^\sharp((\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{y} \doteq \mathbf{x}_3), \mathbf{x}_3 \doteq a(\mathbf{x}_2)) &= [\![\mathbf{x}_1 := \mathbf{x}_3]\!]^\sharp(\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \\
&= (\mathbf{x}_1 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{x}_3 \doteq a(\mathbf{x}_2))\,. \qquad \square
\end{aligned}$$

In order to precisely infer *all* valid Herbrand equalities, we observe:

Proposition 3. *1.* $[\![\mathbf{x}_j := t]\!]^\sharp$ *and* $[\![\mathbf{x}_j := ?]\!]^\sharp$ *preserve* false *and commute with "$\sqcup$".*
2. In the first argument, $[\![\mathsf{call}]\!]^\sharp$ *maps* true *to* false *and translates "$\wedge$"into "$\sqcup$", i.e.,*

$$[\![\mathsf{call}]\!]^\sharp(\mathsf{true}, \phi) = \mathsf{false} \quad \textit{and} \quad [\![\mathsf{call}]\!]^\sharp(\phi_1 \wedge \phi_2, \phi) = [\![\mathsf{call}]\!]^\sharp(\phi_1, \phi) \sqcup [\![\mathsf{call}]\!]^\sharp(\phi_2, \phi)\,.$$

In the second argument, $[\![\mathsf{call}]\!]^\sharp$ *preserves* false *and commutes with "$\sqcup$", i.e.,*

$$[\![\mathsf{call}]\!]^\sharp(\phi, \mathsf{false}) = \mathsf{false} \quad \textit{and} \quad [\![\mathsf{call}]\!]^\sharp(\phi, \phi_1 \sqcup \phi_2) = [\![\mathsf{call}]\!]^\sharp(\phi, \phi_1) \sqcup [\![\mathsf{call}]\!]^\sharp(\phi, \phi_2)\,.$$

Proof: Statement 1 easily follows from the definitions. Therefore we only prove the second statement about the properties of $[\![\mathsf{call}]\!]^\sharp$. The assertion concerning the second argument easily follows from assertion 1. The assertion about the transformation of true in the first argument follows from the definition. Therefore, it remains to consider a conjunction $\phi_1 \wedge \phi_2$ in the first argument of $[\![\mathsf{call}]\!]^\sharp$. We distinguish two cases.

Case 1: $\phi_1 \wedge \phi_2$ is not satisfiable, i.e., equivalent to false. Then $[\![\mathsf{call}]\!]^\sharp(\phi_1 \wedge \phi_2, \phi) = [\![\mathbf{x}_1 := ?]\!]^\sharp\,\phi$. If any of the ϕ_i is also not satisfiable, then $[\![\mathsf{call}]\!]^\sharp(\phi_i, \phi) = [\![\mathbf{x}_1 := ?]\!]^\sharp\,\phi$ which subsumes the effect of any assignment $\mathbf{x}_1 := e$ onto ϕ, and the assertion follows. Therefore assume that both ϕ_1 and ϕ_2 are satisfiable. Each of them then can be written as $\phi_i' \wedge (\mathbf{y} \doteq e_i)$. If any of the ϕ_i' is not implied by ϕ, then again $[\![\mathsf{call}]\!]^\sharp(\phi_i, \phi) = [\![\mathbf{x}_1 := ?]\!]^\sharp\,\phi$ which subsumes the effect of the assignment $\mathbf{x}_1 := e_{3-i}$ onto ϕ. Thus,

$$[\![\mathsf{call}]\!]^\sharp(\phi_1, \phi) \sqcup [\![\mathsf{call}]\!]^\sharp(\phi_2, \phi) \;=\; [\![\mathsf{call}]\!]^\sharp(\phi_i, \phi) \;=\; [\![\mathbf{x}_1 := ?]\!]\;\phi \;=\; [\![\mathsf{call}]\!]^\sharp(\phi_1 \wedge \phi_2, \phi)\,.$$

If on the other hand, both ϕ_i' are implied by ϕ, then $\phi_1' \wedge \phi_2'$ is satisfiable. Thus, $\sigma(e_1) \neq \sigma(e_2)$ for any $\sigma \models \phi_1' \wedge \phi_2'$. In particular, $e_1 \doteq e_2$ cannot be implied by ϕ. Since ϕ_i' is implied by ϕ, $[\![\mathsf{call}]\!]^\sharp(\phi_i, \phi) = [\![\mathbf{x}_1 := e_i]\!]^\sharp\,\phi$. On the other hand, for every ψ containing $\mathbf{x}_1$, it is impossible that both $\phi \Rightarrow \psi[e_1/\mathbf{x}_1]$ and $\phi \Rightarrow \psi[e_2/\mathbf{x}_1]$ hold. Therefore, the least upper bound of $[\![\mathsf{call}]\!]^\sharp(\phi_1, \phi)$ and $[\![\mathsf{call}]\!]^\sharp(\phi_2, \phi)$ is given by the conjunction of all ψ implied by ϕ which do not contain $\mathbf{x}_1$. This conjunction precisely equals $[\![\mathbf{x}_1 := ?]\!]^\sharp\;\phi = [\![\mathsf{call}]\!]^\sharp(\mathsf{false}, \phi)$, and the assertion follows.
Case 2: $\phi_1 \wedge \phi_2$ is satisfiable. Then also both of the ϕ_i are satisfiable and can be written as conjunctions $\phi_i' \wedge (\mathbf{y} \doteq e_i)$ for some ϕ_i' and e_i not containing $\mathbf{y}$. If ϕ does not imply $\phi_1' \wedge \phi_2'$, then both sides of the equation are equal to $[\![\mathbf{x}_1 := ?]\!]\;\phi$ and nothing is to prove. Therefore, assume that $\phi \Rightarrow \phi_1' \wedge \phi_2'$. If ϕ also implies $e_1 \doteq e_2$, then for every ψ, $\phi \Rightarrow \psi[e_1/\mathbf{x}_1]$ iff $\phi \Rightarrow \psi[e_2/\mathbf{x}_1]$. Therefore in this case,

$$[\![\mathsf{call}]\!]^\sharp(\phi_i, \phi) \;=\; [\![\mathbf{x}_1 := e_1]\!]^\sharp\;\phi \;=\; [\![\mathbf{x}_1 := e_2]\!]^\sharp\;\phi \;=\; [\![\mathsf{call}]\!]^\sharp(\phi_1 \wedge \phi_2, \phi)$$

and the assertion follows. If ϕ does not imply $e_1 \doteq e_2$, the least upper bound of $[\![\mathbf{x}_1 := e_i]\!]^\sharp\;\phi$ is the conjunction of all ψ not containing $\mathbf{x}_1$ which are implied by ϕ — which equals:

$$[\![\mathbf{x}_1 := ?]\!]^\sharp\;\phi \;=\; [\![\mathsf{call}]\!]^\sharp(\phi' \wedge (\mathbf{y} \doteq e_1), \phi) \;=\; [\![\mathsf{call}]\!]^\sharp(\phi_1 \wedge \phi_2, \phi)$$

for $\phi' \equiv \phi_1' \wedge \phi_2' \wedge (e_1 \doteq e_2)$, and the assertion follows. □

Applying the abstraction $\alpha_{\mathbf{R}}$ to the constraint system $\mathbf{R}$ of reaching states, we obtain the constraint system $\mathbf{H}$:

$$\begin{array}{lllll}
[\mathbf{H}1] & \mathbf{H}(\mathbf{Main}) & \Leftarrow \mathsf{true} & & \\
[\mathbf{H}2] & \mathbf{H}(f) & \Leftarrow \mathbf{H}(u)\,, & & \text{if } (u, f, _) \in \mathsf{Call} \\
[\mathbf{H}3] & \mathbf{H}(\mathsf{st}_f) & \Leftarrow \mathbf{H}(f) & & \\
[\mathbf{H}4] & \mathbf{H}(v) & \Leftarrow [\![s]\!]^\sharp(\mathbf{H}(u))\,, & & \text{if } (u, s, v) \in \mathsf{Base} \\
[\mathbf{H}5] & \mathbf{H}(v) & \Leftarrow [\![\mathsf{call}]\!]^\sharp(\mathbf{WP}(f), \mathbf{H}(u))\,, & & \text{if } (u, f, v) \in \mathsf{Call}
\end{array}$$

Note that $\mathbf{WP}(f)$ is used in constraint $\mathbf{H}5$ as a summary information for function f. Note also that $\mathbf{H}$ specifies a forwards analysis while $\mathbf{WP}$ accumulates information in a backwards manner. Again by Knaster-Tarski fixpoint theorem, the constraint system $\mathbf{H}$ has a least solution which we denote with $\mathbf{H}(f), \mathbf{H}(u)$, $f \in \mathsf{Funct}, u \in N$. By Proposition 3, we have:

1. $\alpha_{\mathbf{R}}(\mathbf{X} \to \mathcal{T}_\Omega) = \mathsf{true}$;
2. $\alpha_{\mathbf{R}}([\![\mathbf{x}_j := t]\!]\;S) = [\![\mathbf{x}_j := t]\!]^\sharp(\alpha_{\mathbf{R}}(S))$;
3. $\alpha_{\mathbf{R}}([\![\mathbf{x}_j := ?]\!]\;S) = [\![\mathbf{x}_j := ?]\!]^\sharp(\alpha_{\mathbf{R}}(S))$;
4. $\alpha_{\mathbf{R}}([\![\mathsf{call}]\!](T, S)) = [\![\mathsf{call}]\!]^\sharp(\alpha_{\mathbf{S}}(T), \alpha_{\mathbf{R}}(S))$.

We finally obtain:

Theorem 2 (Soundness and Completeness for Side-effect-free Functions). *Assume p is a Herbrand program of size n with k variables.*

1. *For every function f,* $\mathbf{H}(f) = \bigsqcup\{\bigwedge_{i=1}^{k} \mathbf{x}_i \doteq \sigma(\mathbf{x}_i) \mid \sigma \in \mathbf{R}(f)\}$*; and for every program point u,* $\mathbf{H}(u) = \bigsqcup\{\bigwedge_{i=1}^{k} \mathbf{x}_i \doteq \sigma(\mathbf{x}_i) \mid \sigma \in \mathbf{R}(u)\}$.
2. *Given the values* $\mathbf{WP}(f)$*,* $f \in \mathsf{Funct}$*, the least solution of the constraint system* $\mathbf{H}$ *can be computed in time* $\mathcal{O}(n \cdot k \cdot \Delta)$ *where* Δ *is the maximal size of a DAG representation of an occurring conjunction.*

By statement 1 of the theorem, our analysis of side-effect-free functions is not only sound, i.e., never returns a wrong result, but *complete*, i.e., we compute for every program point u and for every function f, the conjunction of *all* equalities which are valid when reaching u and a call of f, respectively. Each application of "$\sqcup$" as well as of any right-hand side in the constraint system $\mathbf{H}$ may at most double the sizes of DAG representations of occurring conjunctions. Together with the corresponding upper bound for the greatest solution of the constraint system $\mathbf{WP}$, the value Δ therefore can be bounded by $2^{\mathcal{O}(n \cdot k)}$. Indeed, this upper bound is *tight* in that it matches the corresponding lower bound for the intra-procedural case [9].

Example 3. Consider again the program from Figure 1. At the start point 0 of $\mathbf{Main}$, no non-trivial equation holds. Therefore, $\mathbf{H}(0) = \mathsf{true}$. For program point 1, we have:

$$\mathbf{H}(1) = [\![\mathbf{x}_3 := a(\mathbf{x}_2)]\!]^\sharp \mathsf{true} = \mathbf{x}_3 \doteq a(\mathbf{x}_2)$$

In Section 4, we have computed the weakest precondition of $\mathbf{y} \doteq \mathbf{x}_1$ for the function f as $(\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{y} \doteq \mathbf{x}_3)$. Since $\mathbf{H}(1)$ implies the equation $\mathbf{x}_3 \doteq a(\mathbf{x}_2)$, we obtain a representation of all equalities valid at program exit 2 by:

$$\begin{aligned} \mathbf{H}(2) &= [\![\mathsf{call}]\!]^\sharp(\mathbf{WP}(f), \mathbf{H}(1)) = [\![\mathbf{x}_1 := \mathbf{x}_3]\!]^\sharp(\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \\ &= (\mathbf{x}_3 \doteq a(\mathbf{x}_2) \wedge (\mathbf{x}_1 \doteq a(\mathbf{x}_2)) \end{aligned}$$

Thus at the return point of $\mathbf{Main}$ both $\mathbf{x}_3 \doteq a(\mathbf{x}_2)$ and $\mathbf{x}_1 \doteq a(\mathbf{x}_2)$ holds. □

6 Programs with Global Variables

In this section, we indicate how our inference algorithm for side-effect-free functions can be extended to an inference algorithm for functions with multiple return values. For the following, we assume that the first m variables are global or, equivalently, that a run of a function f simultaneously computes new values for all variables $\mathbf{x}_1, \ldots, \mathbf{x}_m$. Thus, a function call is now denoted by the vector assignment: $(\mathbf{x}_1, \ldots, \mathbf{x}_m) := f(\mathbf{x}_1, \ldots, \mathbf{x}_k)$. One execution of a function is modeled by a tuple $\tau = (e_1, \ldots, e_m)$ where $e_j \in \mathcal{T}_\Omega(\mathbf{X})$ expresses how the value of variable $\mathbf{x}_j$ after the call depends on the values of the variables before the call. This tuple can also be viewed as a substitution $\tau : \{\mathbf{x}_1, \ldots, \mathbf{x}_m\} \to \mathcal{T}_\Omega(\mathbf{X})$. Accordingly, we change the concrete semantics of a call to:

$$\begin{aligned} [\![\mathsf{call}]\!](T, S) &= \{\sigma[\mathbf{x}_1 \mapsto \sigma(e_1), \ldots, \mathbf{x}_m \mapsto \sigma(e_m)] \mid (e_1, \ldots, e_m) \in T, \sigma \in S\} \\ [\![\mathsf{call}]\!](T_1, T_2) &= \{\tau_1 \circ \tau_2 \mid \tau_i \in T_i\} \end{aligned}$$

In order to obtain effective approximations of the set of effects of function calls, we conceptually abstract one function call computing the values of m variables, by m function calls each of which computes the value of one global variable independently of the others. Technically, we abstract sets of k-tuples with k-tuples of sets. This means that we perform for each variable $x_j \in \{x_1, \ldots, x_m\}$ a separate analysis $\mathbf{P}_j$ of the function body. Accordingly, we generalize the system $\mathbf{WP}$ to a constraint system $\mathbf{P}$:

$$
\begin{array}{lll}
[\mathbf{P}_{\mathrm{j}}1] & \mathbf{P}_j(f) & \Rightarrow \mathbf{P}_j(\mathsf{st}_f) \\
[\mathbf{P}_{\mathrm{j}}2] & \mathbf{P}_j(\mathsf{ret}_f) & \Rightarrow (\mathbf{y}_j \doteq \mathbf{x}_j) \\
[\mathbf{P}_{\mathrm{j}}3] & \mathbf{P}_j(u) & \Rightarrow [\![s]\!]^\sharp(\mathbf{P}_j(v))\,, \qquad \text{if } (u,s,v) \in \mathsf{Base} \\
[\mathbf{P}_{\mathrm{j}}4] & \mathbf{P}_j(u) & \Rightarrow [\![\mathsf{call}_m]\!]^\sharp(\mathbf{P}_1(f),\ldots,\mathbf{P}_m(f),\mathbf{P}_j(v))\,, \text{ if } (u,f,v) \in \mathsf{Call}
\end{array}
$$

Here for $j = 1,\ldots,m$, $\mathbf{y}_j$ is a distinct fresh variable meant to receive the return value for the global variable $\mathbf{x}_j$. The key difference to the constraint system $\mathbf{WP}$ is the treatment of calls by means of the new operator $[\![\mathsf{call}_m]\!]^\sharp$. This operator takes $m+1$ arguments $\phi_1,\ldots,\phi_m,\psi$ (instead of 2 in Section 4). For $j = 1,\ldots,m$, the formula $\phi_j \in \mathbb{E}_{\mathbf{y}_j}$ represents a precondition of the call for the equality $\mathbf{x}_j \doteq \mathbf{y}_j$. The formula ψ on the other hand represents a postcondition for the call. We define:

$$
\begin{array}{ll}
[\![\mathsf{call}_m]\!]^\sharp(\ldots,\mathsf{true},\ldots,\psi) & = \mathsf{true} \\
[\![\mathsf{call}_m]\!]^\sharp(\phi'_1 \wedge (\mathbf{y}_1 \doteq e_1),\ldots,\phi'_m \wedge (\mathbf{y}_m \doteq e_m),\psi) & = \bigwedge_{i\in I} \phi'_i \wedge \psi[e_1/\mathbf{x}_1,\ldots,e_m/\mathbf{x}_m]
\end{array}
$$

where $I = \{i \in \{1,\ldots,m\} \mid \mathbf{x}_i \text{ occurs in } \psi\}$. As in Section 4, $\phi_j \Leftrightarrow \mathsf{true}$ implies that the set of effects is empty. In this case, the operator returns true. Therefore, now assume that for every j, ϕ_j is equivalent to $\phi'_j \wedge \mathbf{y}_j \doteq e_j$ where ϕ'_j and e_j contain only variables from $\mathbf{X}$. If for all j, ϕ'_j equals true, i.e., the return value for $\mathbf{x}_j$ equals e_j, then the call behaves like the substitution $\psi[e_1/\mathbf{x}_1,\ldots,e_m/\mathbf{x}_m]$, i.e., the multiple assignment $(x_1,\ldots,x_m) := (e_1,\ldots,e_m)$. Otherwise, we add the preconditions ϕ'_i for every $\mathbf{x}_i$ occurring in ψ to guarantee that all return values for $\mathbf{x}_i$ are equal to e_i.

As in Section 5, we can use the greatest solution of $\mathbf{P}$ to construct a constraint system $\mathbf{H}'$ from $\mathbf{H}$ by replacing the constraints $\mathbf{H}5$ for calls with the new constraints:

$$
[\mathbf{H}5'] \quad \mathbf{H}(v) \Leftarrow [\![\mathsf{call}_m]\!]^\sharp(\mathbf{P}_1(f),\ldots,\mathbf{P}_m(f),\mathbf{H}(u))\,, \text{ if } (u,f,v) \in \mathsf{Call}
$$

Here, the necessary new abstract operator $[\![\mathsf{call}_m]\!]^\sharp$ for calls is defined by:

$$
\begin{array}{ll}
[\![\mathsf{call}_m]\!]^\sharp(\ldots,\mathsf{true},\ldots,\psi) & = \mathsf{false} \\
[\![\mathsf{call}_m]\!]^\sharp(\phi'_1 \wedge (\mathbf{y}_1 \doteq e_1),\ldots,\phi'_m \wedge (\mathbf{y}_m \doteq e_m),\psi) = & \\
\qquad \exists^\sharp \mathbf{y}_1,\ldots,\mathbf{y}_m.\psi[\mathbf{y}/\mathbf{x}] \wedge \bigwedge_{j\in I}(\mathbf{x}_j \doteq e_j[\mathbf{y}/\mathbf{x}]) &
\end{array}
$$

where $[\mathbf{y}/\mathbf{x}]$ is an abbreviation for the replacement $[\mathbf{y}_1/\mathbf{x}_1,\ldots,\mathbf{y}_m/\mathbf{x}_m]$ and I denotes the set $\{i \mid \psi \Rightarrow \phi'_i\}$. We find:

Theorem 3 (Soundness). *Assume we are given a Herbrand program p with m globals.*

1. *The greatest solution of the constraint system $\mathbf{P}$ for p yields for every function f of p, safe preconditions for the postconditions $\mathbf{x}_i \doteq \mathbf{y}_i$, $i = 1,\ldots,m$.*
2. *The least solution of the constraint system $\mathbf{H}'$ for p yields for every program point u of p, a conjunction of Herbrand equalities which are valid at u.*

The analysis has running-time $\mathcal{O}(n \cdot m^2 \cdot k \cdot \Delta)$ where n is the size of the program and Δ is the maximal size of a conjunction occurring during the analysis. □

At each evaluation of a constraint during the fixpoint computation for $\mathbf{P}$ the maximal size of a conjunction is at most multiplied by a factor of $(m+1)$. Since the number of such evaluations is bounded by $\mathcal{O}(n \cdot m \cdot k)$, we conclude that Δ is bounded by $(m+1)^{\mathcal{O}(n\cdot m\cdot k)}$. Beyond mere soundness, we can say more about the quality of our analysis. In fact, it is strong enough to determine all interprocedural *Herbrand constants*, i.e., to infer for all program points, all equalities of the form $\mathbf{x}_j \doteq t$, t a ground term.

Theorem 4 (Completeness for Constants). *Assume p is a Herbrand program of size n with m globals. Then the following holds:*

1. *For every program point u of p, every variable $\mathbf{x}_j \in \mathbf{X}$ and ground term t, the equality $\mathbf{x}_j \doteq t$ holds at u iff it is implied by $\mathbf{H}_m(u)$.*
2. *All Herbrand constants up to size d can be determined in time $\mathcal{O}(n \cdot m^2 \cdot k^2 \cdot d)$.*

Thus, our algorithm allows for maximally precise interprocedural propagation of Herbrand constants. Moreover, if we are interested in constants up to a given size only, the algorithm can be tuned to run in polynomial time.

Proof: [Sketch] The idea for a proof of the first assertion of Theorem 4 is to introduce a new liberal notion of effect of a function which describes the effect by means of a tuple of sets (instead of a set of tuples). Similar to Sections 4 and 5 one then proves that the constraint systems $\mathbf{P}$ together with $\mathbf{H}_m$ precisely compute all Herbrand equalities valid relative to the liberal notion of effect. This implies that our analysis is *sound*. In order to prove that it is *complete* for equalities $\mathbf{x}_j \doteq t$, t a ground term, we show that if two states at a program point u computed with the liberal effect result in different values for $\mathbf{x}_j$ then there are also two states at u computed according to the strict notion of effects which differ in their values for $\mathbf{x}_j$. □

7 Conclusion

We have presented an interprocedural algorithm for inferring valid Herbrand equalities. Our analysis is *complete* for side-effect-free functions in that it allows us to infer *all* valid Herbrand equalities. We also indicated that our analysis for procedures with more than one global still allows us to determine *all* Herbrand constants. Constant propagation can even be tuned to run in polynomial time if we are interested in constants of bounded size only. Our key idea for the case of side-effect-free functions is to describe the effect of a function by its weakest precondition of the equality $\mathbf{y} \doteq \mathbf{x}_1$.

It remains for future work to investigate the practical usability of the proposed analysis. It also might be interesting to see whether other interprocedural analyses can take advantage of a related approach. In [16], for instance, we discuss an application for determining affine relations. In [15] we have presented an analysis of Herbrand equalities which takes disequality guards into account. It is completely open in how far this intra-procedural analysis can be generalized to some inter-procedural setting.

References

1. B. Alpern, M. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. In *15th ACM Symp. on Principles of Programming Languages (POPL)*, pages 1–11, 1988.
2. K. R. Apt and G. D. Plotkin. Countable Nondeterminism and Random Assignment. *Journal of the ACM*, 33(4):724–767, 1986.
3. C. Click and K. D. Cooper. Combining Analyses, Combining Optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, 1995.
4. J. Cocke and J. T. Schwartz. Programming languages and their compilers. Courant Institute of Mathematical Sciences, NY, 1970.

5. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Electronic Notes in Theoretical Computer Science*, 6, 1997. URL: `www.elsevier.nl/locate/entcs/volume6.html`.
6. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th ACM Symp. on Principles of Programming Languages (POPL)*, 1977.
7. P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Recursive Procedures. In E. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts*, pages 237–277. North-Holland, 1977.
8. K. Gargi. A Sparse Algorithm for Predicated Global Value Numbering. In *ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 45–56, 2002.
9. S. Gulwani and G. C. Necula. A Polynomial-time Algorithm for Global Value Numbering. In *11th Int. Static Analysis Symposium (SAS),*. Springer Verlag, 2004.
10. S. Gulwani and G. C. Necula. Global Value Numbering Using Random Interpretation. In *31st ACM Symp. on Principles of Programming Languages (POPL)*, pages 342–352, 2004.
11. G. A. Kildall. A Unified Approach to Global Program Optimization. In *First ACM Symp. on Principles of Programming Languages (POPL)*, pages 194–206, 1973.
12. J. Knoop, O. Rüthing, and B. Steffen. Code Motion and Code Placement: Just Synonyms? In *6th ESOP*, LNCS 1381, pages 154–196. Springer-Verlag, 1998.
13. J. Knoop and B. Steffen. The Interprocedural Coincidence Theorem. In *Compiler Construction (CC)*, pages 125–140. LNCS 541, Springer-Verlag, 1992.
14. S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice Hall, Engelwood Cliffs, New Jersey, 1981.
15. M. Müller-Olm, O. Rüthing, and H. Seidl. Checking Herbrand Equalities and Beyond. In *Proceedings of VMCAI 2005*. to appear, Springer-Verlag, 2005.
16. M. Müller-Olm, H. Seidl, and B. Steffen. Interprocedural Analysis for Free. Technical Report 790, Fachbereich Informatik, Universität Dortmund, 2004.
17. J. H. Reif and R. Lewis. Symbolic Evaluation and the Gobal Value Graph. In *4th ACM Symp. on Principles of Programming Languages (POPL)*, pages 104–118, 1977.
18. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *15th ACM Symp. on Principles of Programming Languages (POPL)*, pages 12–27, 1988.
19. O. Rüthing, J. Knoop, and B. Steffen. Detecting Equalities of Variables: Combining Efficiency with Precision. In *6th Int. Static Analysis Symposium (SAS)*, LNCS 1694, pages 232–247. Springer-Verlag, 1999.
20. M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In [14], chapter 7, pages 189–233.
21. B. Steffen. Optimal Run Time Optimization—Proved by a New Look at Abstract Interpretations. In *Proc. 2nd International Joint Conference on Theory and Practice of Software Development (TAPSOFT'87)*, LNCS 249, pages 52–68. Springer Verlag, 1987.
22. B. Steffen, J. Knoop, and O. Rüthing. The Value Flow Graph: A Program Representation for Optimal Program Transformations. In *3rd European Symp. on Programming (ESOP)*, LNCS 432, pages 389–405. Springer-Verlag, 1990.
23. B. Steffen, J. Knoop, and O. Rüthing. Efficient Code Motion and an Adaption to Strength Reduction. In *4th International Joint Conference on the Theory and Practice of Software Development* (*TAPSOFT*), LNCS 494, pages 394–415. Springer-Verlag, 1991.

Analysis of Modular Arithmetic

Markus Müller-Olm[1] and Helmut Seidl[2]

[1] Universität Dortmund, Fachbereich Informatik, LS 5,
Baroper Str. 301, 44221 Dortmund, Germany
markus.mueller-olm@cs.uni-dortmund.de
[2] TU München, Institut für Informatik, I2,
80333 München, Germany
seidl@in.tum.de

Abstract. We consider integer arithmetic modulo a power of 2 as provided by mainstream programming languages like Java or standard implementations of C. The difficulty here is that the ring $\mathbb{Z}_m$ of integers modulo $m = 2^w$, $w > 1$, has zero divisors and thus cannot be embedded into a field. Not withstanding that, we present intra- and inter-procedural algorithms for inferring for every program point u, affine relations between program variables valid at u. Our algorithms are not only sound but also *complete* in that they detect *all* valid affine relations. Moreover, they run in time linear in the program size and polynomial in the number of program variables and can be implemented by using the same modular integer arithmetic as the target language to be analyzed.

1 Introduction

Analyses for automatically finding linear invariants in programs have been studied for a long time [6, 3, 4, 7, 12, 11, 9]. With the notable exception of Granger [3], none of these analyses can find out, that the linear invariant $21 \cdot \mathbf{x} - \mathbf{y} = 1$ holds upon termination of the following Java program:

```
class Eins {
    public static void main(String [] argv) {
        int x = 1022611261; int y = 0;
        if (argv.length > 0) {
            x = 1; y = 20;
        }
        System.out.println("" + (21*x-y));
    }
}
```

Why is this? In order to allow implementing arithmetic operations by the efficient instructions provided by processors, Java, like other common programming languages, performs arithmetic operations for integer types modulo $m = 2^w$ where $w = 32$, if the result expression is of type `int`, and $w = 64$, if the result expression is of type `long` [2–p. 32]. The invariant $21 \cdot \mathbf{x} - \mathbf{y} = 1$ is valid because $21 * 1022611261 = 1$ modulo 2^{32}. In order to work with mathematical structures

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 46–60, 2005.

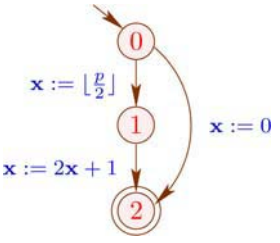


Fig. 1. $\mathbb{Z}_p$ interpretation is unsound

with nice properties analyses for finding linear invariants typically interpret variables by members from a field, e.g., the set $\mathbb{Q}$ of rational numbers [6, 11, 8], or $\mathbb{Z}_p = \mathbb{Z}/(p\mathbb{Z})$ for prime numbers p [4]. Even worse: analyses based on $\mathbb{Z}_p$ for a fixed prime p alone may yield unsound results.[1] In the small flow graph in Fig. 1, for instance, x is a constant at program point 2 if variables take values in $\mathbb{Z}_p$ for a prime number $p > 2$, but it is not a constant if variables take values in $\mathbb{Z}_m$. Interestingly, the given problem is resolved by Granger's analysis which not only detects all affine relations between integer variables but also all affine *congruence* relations. On the other hand, Granger's analysis is just intra-procedural, and no upper complexity bound to his algorithm is known.

In this paper we present intra- and inter-procedural analyses that are sound and, up to the common abstraction of guarded to non-deterministic branching, complete with respect to arithmetic modulo powers of 2. Our analyses are thus tightly tailored for the arithmetic used in mainstream programming languages. For this arithmetic, our analyses are more precise than analyses based on computing over $\mathbb{Q}$, or $\mathbb{Z}_p$ and, in contrast to analyses based on computing over $\mathbb{Z}_p$ with a fixed prime p, they are *sound* w.r.t. the arithmetic used in mainstream programming languages. Technically, our new analyses are based on the methods from linear algebra that we have studied previously [8, 11]. The major new difficulty is that unlike $\mathbb{Q}$ and $\mathbb{Z}_p$, $\mathbb{Z}_m$ is no longer a field. In particular, $\mathbb{Z}_m$ has zero divisors implying that not every non-zero element is invertible. Therefore, results from linear algebra over fields do not apply to sets of vectors and matrices over $\mathbb{Z}_m$. However, these sets are still *modules* over $\mathbb{Z}_m$. An extensive account of linear algebra techniques for modules over abstract rings can, e.g., be found in [5, 13]. Here, we simplify the general techniques to establish the properties of $\mathbb{Z}_m$ which suffice to implement similar algorithms as in [8, 11]. Interestingly, the new analyses provide extra useful information beyond analyses over $\mathbb{Q}$ alone, for instance, whether or not a variable is always a multiple of 2.

Besides the soundness and completeness issues discussed above, there is another advantage of our analyses that is perhaps more important from a practical point of view than precision. For any algorithm based on computing in $\mathbb{Q}$, we must use some representation for rational numbers. When using floating point

[1] If the primes of the analysis are chosen randomly, the resulting analysis is at least "probabilistically sound" [4].

numbers, we must cope with rounding errors and numerical instability. Alternatively, we may represent rational numbers as pairs of integers. Then we can either rely on integers of bounded size as provided by the host language. In this case we must cope with overflows. Or we represent integers by arbitrarily long bit strings. In this case the sizes of our representations may explode. On the other hand, when computing over $\mathbb{Z}_p$, p a prime, special care is needed to get the analysis right. The algorithms proposed in this paper, however, can be implemented using the modulo arithmetic provided by the host language itself. In particular, without any additional effort this totally prevents explosion of number representations, rounding errors, and numerical instability.

Our paper is organized as follows. In Section 2, we investigate the properties of the ring $\mathbb{Z}_m$ for powers $m = 2^w$ and provide basic technology for dealing with generating systems of $\mathbb{Z}_m$-modules. In particular we show how to compute a (finite description of) all solutions of a system of linear equations over $\mathbb{Z}_m$. In Section 3, we show how these insights can be used to construct sound and complete program analyses. We introduce our basic notion of programs together with their concrete semantics. We introduce affine relations and adapt the basic technology provided in [11] for fields to work for rings $\mathbb{Z}_m$. Finally, we sketch how to obtain highly efficient *intraprocedural* analyses of affine relations over $\mathbb{Z}_m$. In Section 4, we summarize and explain further directions of research.

2 The Ring $\mathbb{Z}_m$ for Powers $m = 2^w$

In [5, 13], efficient methods are developed for computing various normal forms of matrices over *principal ideal rings* (PIR's). Here, we are interested in the residue class ring $\mathbb{Z}_m$ for prime powers m which is a special case of a PIR. Accordingly, the general methods from [5, 13] are applicable. It turns out, however, that for prime powers m, the ring $\mathbb{Z}_m$ has a very special structure. In this section, we show how this structure can be exploited to obtain specialized algorithms in which the computation of (generalized) gcd's is (mostly) abandoned. Since the abstract values of our program analyses will be *submodules* of $\mathbb{Z}_m^N$ for suitable N, we also compute the exact maximal length of a strictly ascending chain of such submodules. Since we need *effective representations* of modules, we provide algorithms for dealing with sets of generators. In particular, we show how to solve homogeneous systems of linear equations over $\mathbb{Z}_m$ without gcd computations. In the sequel, let $m = 2^w$, $w \geq 1$. We begin with the following observation.

Lemma 1. *Assume $a \in \mathbb{Z}_m$ is different from 0. Then we have:*

1. *If a is even, then a is a zero divisor, i.e., $a \cdot b = 0$ (modulo m) for some $b \in \mathbb{Z}_m$ different from 0.*
2. *If a is odd, then a is invertible, i.e., $a \cdot b = 1$ modulo m for some $b \in \mathbb{Z}_m$. Using arithmetic modulo m, the inverse b can be computed in time $\mathcal{O}(w)$.*

Proof. Assume $a = 2 \cdot a'$. Then $a \cdot 2^{w-1} = 2^w \cdot a' = 0$ (modulo m). If, on the other hand, a is odd, then a and m are relative prime. Therefore, we can use the

Euclidean algorithm to determine integers x and y such that $1 = a \cdot x + m \cdot y$. Accordingly, $b = x$ (modulo m) is the inverse of a. This algorithm, however, cannot be executed modulo m. In the case where $w = 1$, we know that $\mathbb{Z}_m$ is in fact the field $\mathbb{Z}_2$. Thus, the inverse of $a \neq 0$ (modulo 2) is given by a. If on the other hand $w > 1$, we can use the Euclidean algorithm to determine odd integers x_1 and y_1 with $1 = a \cdot x_1 + 2^{w-1} \cdot y_1$. By computing the square of both sides of this equation, we obtain:

$$1 = a^2 x_1^2 + 2 \cdot a x_1 2^{w-1} y_1 + 2^{(w-1)2} y_1^2$$

Every summand of the right-hand side except the first contains 2^w as a factor and thus equals 0 (modulo m). Hence, $b = a x_1^2$ (modulo m). Since the Euclidean algorithm uses $\mathcal{O}(\log(m))$ operations, the complexity statement follows. □

Example 1. Consider $w = 32$ and $a = 21$. We use the familiar notation of Java int values as elements in the range $[-2^{31}, 2^{31} - 1]$. The Euclidean algorithm applied to a and $m' = 2^{31}$ (or: -2^{31} in signed notation) gives us $x_1 = -1124872387$ and $y_1 = 11$. Then $b = 21 \cdot x_1^2 = 1022611261$ modulo 2^{32} is the inverse of a. □

Since computing inverses can be rather expensive, we will avoid these whenever possible. For $a \in \mathbb{Z}_m$, we define the *rank* of a as $r \in \{0, \ldots, w\}$ iff $a = 2^r \cdot a'$ for some invertible element a'. In particular, the rank is 0 iff a itself is invertible, and the rank is w iff $a = 0$ (modulo m). Note that the rank of a can be computed by determining the length of suffix of zeros in the bit representation of a. If there is no hardware support for this operation, it can be computed with $\mathcal{O}(\log(w))$ arithmetic operations using a variant of binary search.

A subset $M \subseteq \mathbb{Z}_m^k$ of vectors[2] $(x_1, \ldots, x_k)^{\mathrm{t}}$ with entries x_i in $\mathbb{Z}_m$ is a $\mathbb{Z}_m$-*module* iff it is closed under vector addition and scalar multiplication with elements from $\mathbb{Z}_m$. A subset $G \subseteq M$ is a *set of generators* of M iff $M = \{\sum_{i=1}^{l} r_i g_i \mid l \geq 0, r_i \in \mathbb{Z}_m, g_i \in G\}$. Then M is *generated* by G and we write $M = \langle G \rangle$.

For a non-zero vector $x = (x_1, \ldots, x_k)^{\mathrm{t}}$, we call i the *leading index* iff $x_i \neq 0$ and $x_{i'} = 0$ for all $i' < i$. In this case, x_i is the *leading entry* of x. A set of non-zero vectors is in *triangular form* iff for all distinct vectors $x, x' \in G$, the leading indices of x and x' are distinct. Every set G in triangular form contains at most k elements. We define the *rank* of a triangular set G of cardinality s as the sum of the ranks of the leading entries of the vectors of G plus $(k - s) \cdot w$ (to account for $k - s$ zero vectors). Note that this deviates from the common notion of the rank of a matrix.

Assume that we are given a set G in triangular form together with a new vector x. Our goal is to construct a set $\bar{G}$ in triangular form generating the same $\mathbb{Z}_m$-module as $G \cup \{x\}$. If x is the zero vector, then we simply can choose $\bar{G} = G$. Otherwise, let i and $2^r d$ (d invertible) denote the leading index and leading entry of x, respectively. We distinguish several cases:

1. The i-th entry of all vectors $x' \in G$ are 0. Then we choose $\bar{G} = G \cup \{x\}$.

[2] The superscript "t" denotes the *transpose* operation which mirrors a matrix at the main diagonal and changes a row vector into a column vector (and vice versa).

2. i is the leading index of some $y \in G$ where the leading entry equals $2^{r'}d'$ (d' invertible).
   (a) If $r' \leq r$, then we compute $x' = d' \cdot x - 2^{r-r'}d \cdot y$. Thus, the i-th entry of x' equals 0, and we proceed with G and x'.
   (b) If $r' > r$, then we construct a new set G' by replacing y with the vector x. Furthermore, we compute $y' = d \cdot y - 2^{r'-r}d' \cdot x$. Thus, the i-th entry of y' equals 0, and we proceed with G' and y'.

Eventually, we arrive at a set $\bar{G}$ having the desired properties. Moreover, either the resulting $\bar{G}$ equals G or the rank of $\bar{G}$ is strictly less than the rank of G.

Overall, computing a triangular set for a given triangular set and a new vector amounts to at most $\mathcal{O}(k)$ computations of ranks together with $\mathcal{O}(k^2)$ arithmetic operations. On the whole, it therefore can be done with $\mathcal{O}(k \cdot (k + \log(w)))$ operations. Accordingly, we obtain the following theorem:

Theorem 1. *1. Every $\mathbb{Z}_m$-module $M \subseteq \mathbb{Z}_m^k$ is generated by some set G of generators of cardinality at most k.*
2. Given a set G' of generators of cardinality n, a set G of cardinality at most k can be computed in time $\mathcal{O}(n \cdot k \cdot (k + \log(w)))$ such that $\langle G \rangle = \langle G' \rangle$.
3. Every strictly increasing chain of $\mathbb{Z}_m$-modules $M_0 \subset M_1 \subset \ldots \subset M_s \subseteq \mathbb{Z}_m^k$, has length $s \leq k \cdot w$.

Proof. The second statement follows from our construction of triangular sets of generators. Starting from the empty set, which is triangular by definition, we successively add the vectors in G' with the procedure described above. The complexity is then estimated by summing up the operations of these n inclusions.

The first statement trivially follows from the second because M is a finite generator of itself. It remains to consider the third statement. Assume that $M_i \subset M_{i+1}$ for $i = 0, \ldots, s-1$. Consider finite sets G_i of generators for M_i. We construct a sequence of triangular sets generating the same modules as follows. G_0' is the triangular set constructed for G_0. For $i > 0$, G_i' is obtained from G_{i-1}' by successively adding the vectors in G_i to the set G_{i-1}'. Since $M_{i-1} \neq M_i$, the triangular set G_{i-1}' is necessarily different from the set G_i' for all $i = 1, \ldots, s$. Therefore, the ranks of the G_i' are strictly decreasing. Since the maximal possible rank is $k \cdot w$ and ranks are non-negative, the third statement follows. □

Example 2. In order to keep the numbers small, we choose here and in the following examples of this section $w = 4$, i.e., $m = 16$. Consider the vectors $x = (2, 6, 9)^{\mathsf{t}}$ and $y = (0, 2, 4)^{\mathsf{t}}$ with leading indices 1 and 2 and both with leading entry 2. Thus, the set $G = \{x, y\}$ is triangular. Let $z = (1, 2, 1)^{\mathsf{t}}$. We want to construct a triangular set of generators equivalent to $G \cup \{z\}$. Since the leading index of z equals 1, we compare the leading entries of x and z. The ranks of the leading entries of x and z are 1 and 0, respectively. Therefore, we exchange x in the generating set with z while continuing with $x' = x - 2 \cdot z = (0, 2, 7)^{\mathsf{t}}$. The leading index of x' has now increased to 2 . Comparing x' with the vector y, we find that the leading entries have identical ranks. Thus, we can subtract a suitable multiple of y to bring the second component of x' to 0 as well. We compute $x'' = x' - 1 \cdot y = (0, 0, 3)^{\mathsf{t}}$. As triangular set we finally return $\bar{G} = \{z, y, x''\}$. □

For a set of generators G being triangular, does not imply being a *minimal* set of generators. For $w = 3$ consider, e.g., the triangular set $G = \{x, y\}$ where $x = (4, 1)^{\mathsf{t}}, y = (0, 2)^{\mathsf{t}}$. Multiplying x with 2 results in: $2 \cdot x = (8, 2)^{\mathsf{t}} = (0, 2)^{\mathsf{t}} = y$. Thus $\{x\}$ generates the same module as G implying that G is not minimal.

It is well-known that the submodules of $\mathbb{Z}_m^k$ are closed under intersection. Ordered by set inclusion they thus form a complete lattice $\mathbf{Sub}(\mathbb{Z}_m^k)$, like the linear subspaces of $\mathbb{F}^k$ for a field $\mathbb{F}$. However, while the height of the lattice of linear subspaces of $\mathbb{F}^k$ is k for dimension reasons, the height of the lattice of submodules of $\mathbb{Z}_m^k$ is precisely $k \cdot w$. By Theorem 1, $k \cdot w$ is an upper bound for the height and it is not hard to actually construct a chain of this length. The least element of $\mathbf{Sub}(\mathbb{Z}_m^k)$ is $\{\mathbf{0}\}$, the greatest element is $\mathbb{Z}_m^k$ itself. The least upper bound of two submodules M_1, M_2 is given by

$$M_1 \sqcup M_2 = \langle M_1 \cup M_2 \rangle = \{m_1 + m_2 \mid m_i \in M_i\}\,.$$

We turn to the computation of the solutions of systems of linear equations in k variables over $\mathbb{Z}_m$. Here, we consider only the case where the number of equations is at most as large as the number of variables. By adding extra equations with all coefficients equal to zero, we can assume that every such system has precisely k equations. Such a system can be denoted as $A\,\mathbf{x} = b$ where A is a square $(k \times k)$-matrix $A = (a_{ij})_{1 \le i,j \le k}$ with entries $a_{ij} \in \mathbb{Z}_m$, $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)^{\mathsf{t}}$ is a column vector of unknowns and $b = (b_1, \ldots, b_k)^{\mathsf{t}}$ is a column vector of elements $b_i \in \mathbb{Z}_m$. Let $\mathbb{L}$ denote the set of all solutions of $A\,\mathbf{x} = b$. Let $\mathbb{L}_0$ denote the set of all solutions of the corresponding *homogeneous system* $A\,\mathbf{x} = \mathbf{0}$ where $\mathbf{0} = (0, \ldots, 0)^{\mathsf{t}}$. It is well-known that, if the system $A\,\mathbf{x} = b$ has at least one solution x, then the set of all its solutions can be obtained from x by adding solutions of the corresponding homogeneous system, i.e.,

$$\mathbb{L} = \{x + y \mid y \in \mathbb{L}_0\}$$

Let us first consider the case where the matrix A is *diagonal*, i.e., $a_{ij} = 0$ for all $i \neq j$. The following lemma deals completely with this case.

Lemma 2. *Assume A is a diagonal $(k \times k)$-matrix over $\mathbb{Z}_m$ where the diagonal elements are given by $a_{ii} = 2^{w_i} d_i$ for invertible d_i ($w_i = w$ means $a_{ii} = 0$).*

1. *$A\,\mathbf{x} = b$ has a solution iff for all i, w_i does not exceed the rank of b_i.*
2. *If $A\,\mathbf{x} = b$ is solvable, then one solution is given by: $x = (x_1, \ldots, x_k)^{\mathsf{t}}$ with $x_i = 2^{w_i' - w_i} \cdot d_i^{-1} b_i'$ where $b_i = 2^{w_i'} b_i'$ for invertible elements b_i'.*
3. *The set of solutions of the homogeneous system $A\,\mathbf{x} = \mathbf{0}$ is the $\mathbb{Z}_m$-module generated from the vectors: $e^{(j)} = (e_{1j}, \ldots, e_{kj})^{\mathsf{t}}, j = 1, \ldots, k$, where $e_{ij} = 2^{w - w_i}$ if $i = j$ and $e_{ij} = 0$ otherwise.* □

In contrast to equation systems over fields, a homogeneous system $A\,\mathbf{x} = \mathbf{0}$ thus may have non-trivial solutions — even if all entries a_{ii} are different from 0. Note also, that in contrast to inhomogeneous systems, sets of generators for homogeneous systems can be computed *without* computing inverses.

Example 3. Let $w = 4$, i.e., $m = 16$, and

$$A = \begin{pmatrix} 2 & 0 \\ 0 & 8 \end{pmatrix}$$

Then the $\mathbb{Z}_m$-module of solutions of $A\,\mathbf{x} = 0$ is generated by the two vectors $e^{(1)} = (8,0)^{\mathsf{t}}$ and $e^{(2)} = (0,2)^{\mathsf{t}}$. □

For the case where the matrix A is not diagonal, we adapt the concept of invertible column and row transformations known from linear algebra to bring A into diagonal form. More precisely, we have:

Lemma 3. *Let A denote an arbitrary $(k \times k)$-matrix over $\mathbb{Z}_m$. Then we have:*

1. *A can be decomposed into matrices: $A = L \cdot D \cdot R$ where D is diagonal and L, R are invertible $(k \times k)$-matrices over $\mathbb{Z}_m$.*
2. *W.r.t. this decomposition, x is a solution of $A\,\mathbf{x} = b$ iff $x = R^{-1}x'$ for a solution x' of the system $D\,\mathbf{x} = b'$ for $b' = L^{-1}b$.*
3. *The matrix D together with the matrix R^{-1} and the vector $b' = L^{-1}\,b$ can be computed in time $\mathcal{O}(\log(w) \cdot k^3)$. In particular, computation of inverses is not needed for the decomposition.*

Proof. In order to prove that every matrix A can indeed be decomposed into a product $A = L \cdot D \cdot R$ for a diagonal matrix D and invertible matrices L, R over $\mathbb{Z}_m$, we recall the corresponding technique over fields from linear algebra. Recall that the idea for fields consisted in successively selecting a non-zero Pivot element (i,j) in the current matrix. Since every non-zero element in a field is invertible, the entry d at (i,j) has an inverse d^{-1}. By multiplying the row with d^{-1}, one can bring the entry (i,j) to 1. Then one can apply column and row transformations to bring all other elements in the same column or row to zero. Finally, by exchanging suitable columns or rows, one can bring the former Pivot entry into the diagonal. In contrast, when computing in the ring $\mathbb{Z}_m$, we do not have inverses for all non-zero elements, and even if there are inverses, we would like to avoid their construction. Therefore, we refine the selection rule for Pivot elements by always selecting as a Pivot element the (i,j) where the entry $d = 2^r d'$ of the current matrix has *minimal rank* r, and d' is invertible over $\mathbb{Z}_m$. Since r has been chosen minimal, still all other elements in row i and column j are multiples of 2^r. Therefore, all these entries can be brought to 0 by multiplying the corresponding row or column with d' and then subtracting a suitable multiple of the i-th row or j-th column, respectively. These elementary transformations are invertible since d' is invertible. Finally, by suitable exchanges of columns or rows, the entry (i,j) can be moved into the diagonal. Proceeding with the classical construction for fields, the inverses of the chosen elementary column transformations are collected in the matrix R while the inverses of the chosen elementary row transformations are collected in the matrix L. Since the elementary transformations which we apply only express exchange of columns or rows, multiplication with an invertible element or adding of a multiple of one column / row to the other, these transformations are also invertible over $\mathbb{Z}_m$.

Now it should be clear how the matrix D together with the matrix R^{-1} and the vector $b' = L^{-1}b$ can be computed. The matrix R^{-1} is obtained by starting from the unit matrix and then performing the same sequence of column operations on it as on A. Also, the vector b' is obtained by performing on b the

same sequence of row transformations as on A. In particular, this provides us with the complexity bound as stated in item (3). □

Putting lemmas 2 and 3 together we obtain:

Theorem 2. *1. A representation of the set $\mathbb{L}_0$ of a homogeneous equation system $A\,\mathbf{x} = 0$ over $\mathbb{Z}_m$ can be computed without resorting to the computation of inverses in time $\mathcal{O}(\log(w) \cdot k^3)$.*
2. A representation of the set $\mathbb{L}$ of all solutions of an equation system $A\,\mathbf{x} = b$ over $\mathbb{Z}_m$ can be computed in time $\mathcal{O}(w \cdot k + \log(w) \cdot k^3)$.

Example 4. Consider, for $w = 4$, i.e., $m = 16$, the equation system with the two equations

$$\begin{aligned} 12\mathbf{x}_1 + 6\mathbf{x}_2 &= 10 \\ 14\mathbf{x}_1 + 4\mathbf{x}_2 &= 8 \end{aligned}$$

We start with

$$A_0 = \begin{pmatrix} 12 & 6 \\ 14 & 4 \end{pmatrix},\ b_0 = \begin{pmatrix} 10 \\ 8 \end{pmatrix},\ R_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

We cannot use $(1,1)$ with entry 12 as a Pivot, since the rank of 12 exceeds the ranks of 14 and 6. Therefore we choose $(1,2)$ with entry 6. We bring the entry at $(2,2)$ to 0 by multiplying the second row with 3 and subtracting the first row twice in A_0 and in b_0:

$$A_1 = \begin{pmatrix} 12 & 6 \\ 2 & 0 \end{pmatrix},\ b_1 = \begin{pmatrix} 10 \\ 4 \end{pmatrix},\ R_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

By subtracting twice the second column from the first in A_1 and R_1, we obtain:

$$A_2 = \begin{pmatrix} 0 & 6 \\ 2 & 0 \end{pmatrix},\ b_2 = \begin{pmatrix} 10 \\ 4 \end{pmatrix},\ R_2 = \begin{pmatrix} 1 & 0 \\ 14 & 1 \end{pmatrix}$$

Now, we exchange the columns 1 and 2 in A_3 and R_3:

$$A_3 = \begin{pmatrix} 6 & 0 \\ 0 & 2 \end{pmatrix},\ b_3 = \begin{pmatrix} 10 \\ 4 \end{pmatrix},\ R_3 = \begin{pmatrix} 0 & 1 \\ 1 & 14 \end{pmatrix}$$

Since $3 \cdot 11 = 1 \bmod 16$, we can easily read off $x'_0 = \binom{11 \cdot 5}{2} = \binom{7}{2}$ as a solution of $A_3\,\mathbf{x} = b_3$. We also see that the two vectors $x'_1 = \binom{8}{0}$ and $x'_1 = \binom{0}{8}$ generate the module of solutions of the homogeneous system $A_3\mathbf{x} = \mathbf{0}$. Consequently, $x_0 = R_3\,x'_0 = \binom{2}{3}$ is a solution of $A_0\,\mathbf{x} = b_0$ and the two vectors $x_1 = R_3\,x'_1 = \binom{0}{8}$ and $x_1 = R_3\,x'_2 = \binom{8}{0}$ generate the module of solutions of the homogeneous system $A_0\,\mathbf{x} = \mathbf{0}$. We conclude that the set of solutions of $A_0\,\mathbf{x} = b_0$ (over $\mathbb{Z}_{16}$) is

$$\mathbb{L} = \left\{ \binom{2+8a}{3+8b} \mid a, b \in \mathbb{Z}_{16} \right\} = \left\{ \binom{2}{3}, \binom{2}{11}, \binom{10}{3}, \binom{10}{11} \right\}$$

□

3 Affine Program Analysis

In the last section, we have proposed algorithms for reducing sets of generators of $\mathbb{Z}_m$-modules and for solving systems of (homogeneous) linear equations over $\mathbb{Z}_m$. In this section, we show how these algorithms can be plugged into the algorithmic skeletons of the sound and complete analyses of affine relations over fields as, e.g., presented in [11] to obtain sound and complete analyses of affine relations over $\mathbb{Z}_m$. For the sake of an easier comparison, we use the same conventions as in

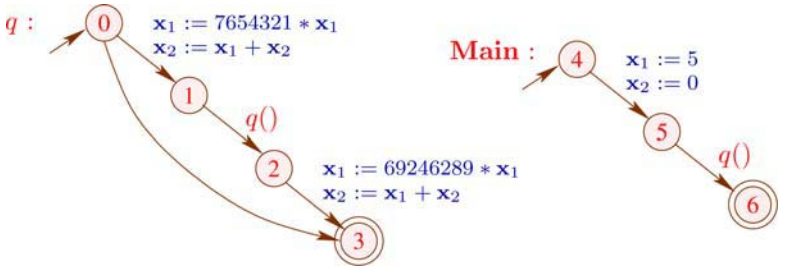


Fig. 2. An inter-procedural program

[11] which we recall here briefly in order to be self-contained. Thus, programs are modeled by systems of non-deterministic flow graphs that can recursively call each other as in Figure 2. Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ be the set of (global) variables the program operates on. We use $\mathbf{x}$ to denote the column vector of variables $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)^{\mathrm{t}}$. In this paper, we assume that the variables take values in the ring $\mathbb{Z}_m$. Thus, a *state* assigning values to the variables is modeled by a k-dimensional (column) vector $x = (x_1, \ldots, x_k)^{\mathrm{t}} \in \mathbb{Z}_m^k$; x_i being the value assigned to variable $\mathbf{x}_i$. For a state x, a variable $\mathbf{x}_i$ and a value $c \in \mathbb{Z}_m$, we write $x[\mathbf{x}_i \mapsto c]$ for the state $(x_1, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_k)^{\mathrm{t}}$ as usual.

We assume that the basic statements in our programs either are *affine assignments* of the form $\mathbf{x}_j := t_0 + \sum_{i=1}^{k} t_i \mathbf{x}_i$ (with $t_i \in \mathbb{Z}_m$ for $i = 0, \ldots, k$ and $\mathbf{x}_j \in \mathbf{X}$) or *non-deterministic assignments* of the form $\mathbf{x}_j := ?$ (with $\mathbf{x}_j \in \mathbf{X}$). We annotated the edges in Fig. 2 with sequences of assignments just in order to reduce the number of program points. Since assignments $\mathbf{x}_j := \mathbf{x}_j$ have no effect onto the program state, they are skip-statements and omitted in pictures. Skip-statements can be used to abstract guards. This amounts to replacing conditional branching in the original program with non-deterministic branching. It is relative to this common abstraction when we say an analysis is complete.

Non-deterministic assignments $\mathbf{x}_j := ?$ can be used as a safe abstraction of statements our analysis cannot handle precisely, for example of assignments $\mathbf{x}_j := t$ with non-affine expressions t or of read statements $\mathsf{read}(\mathbf{x}_j)$.

In this setting, a *program* comprises a finite set Proc of *procedure names* together with one distinguished procedure **Main**. Execution starts with a call to **Main**. Each procedure $q \in \mathsf{Proc}$ is specified by a distinct edge-labeled *control flow graph* with a single start point st_q and a single return point ret_q where each edge is either labeled with an assignment or a call to some procedure.

The key idea of [11] which we take up here for the analysis of modular arithmetic, is to construct a precise abstract interpretation of a constraint system characterizing the program executions that reach program points. For that, program executions or *runs* are represented by sequences r of affine assignments:

$$r \equiv s_1; \ldots; s_m$$

where s_i are assignments of the form $\mathbf{x}_j := t$, $\mathbf{x}_j \in \mathbf{X}$ and $t \equiv t_0 + \sum_{i=1}^{k} t_i \mathbf{x}_i$ for some $t_0, \ldots, t_k \in \mathbb{Z}_m$. (Non-deterministic assignments give rise to multiple

runs.) We write Runs for the set of runs. Every assignment statement $\mathbf{x}_i := t$ induces a state transformation $[\![\mathbf{x}_j := t]\!] : \mathbb{Z}_m^k \to \mathbb{Z}_m^k$ given by

$$[\![\mathbf{x}_j := t]\!]x \;=\; x[\mathbf{x}_j \mapsto t(x)]\,,$$

where $t(x)$ is the value of term t in state x. This definition is inductively extended to runs: $[\![\varepsilon]\!] = \mathsf{Id}$, where Id is the identical mapping and $[\![ra]\!] = [\![a]\!] \circ [\![r]\!]$.

A closer look reveals that the state transformation of an affine assignment $\mathbf{x}_j := t_0 + \sum_{i=1}^k t_i\mathbf{x}_i$ is in fact an affine transformation. As a composition of affine transformations, the state transformer of a run is therefore also an affine transformation — no matter whether we compute over fields or some $\mathbb{Z}_m$. Hence, for any run r, we can choose $A_r \in \mathbb{Z}_m^{k^2}$ and $b_r \in \mathbb{Z}_m^k$ such that $[\![r]\!]x = A_r x + b_r$.

The definition of *affine relations* over $\mathbb{Z}_m$ is completely analogous to affine relations over fields. So, an *affine relation* over $\mathbb{Z}_m^k$ is an equation $a_0 + a_1\mathbf{x}_1 + \ldots + a_k\mathbf{x}_k = 0$ for some $a_i \in \mathbb{Z}_m$. As for fields, we represent such a relation by the column vector $a = (a_0, \ldots, a_k)^{\mathrm{t}}$. Instead of a vector space, the set of all affine relations now forms a $\mathbb{Z}_m$-module isomorphic to $\mathbb{Z}_m^{k+1}$. We say that the vector $y \in \mathbb{Z}_m^k$ *satisfies* the affine relation a iff $a_0 + a' \cdot y = 0$ where $a' = (a_1, \ldots, a_k)^{\mathrm{t}}$ and "$\cdot$" denotes the scalar product. This fact is denoted by $y \models a$.

We say that the affine relation a is *valid* after a single run r iff $[\![r]\!]x \models a$ for all $x \in \mathbb{Z}_m^k$, i.e., iff $a_0 + a' \cdot [\![r]\!]x = 0$ for all $x \in \mathbb{Z}_m^k$; x represents the unknown initial state. Thus, $a_0 + a' \cdot [\![r]\!]\mathbf{x} = 0$ is the *weakest precondition* for validity of the affine relation a after run r. In [11], we have shown in the case of fields, that the weakest precondition can be computed by a *linear transformation* applied to the vector a. The very same argumentation works as well in the more general case of arbitrary rings. More specifically, this linear transformation is given by the following $(k+1)^2$ matrix W_r:

$$W_r = \left(\begin{array}{c|c} 1 & b_r^{\mathrm{t}} \\ \hline 0 & A_r^{\mathrm{t}} \end{array}\right) \tag{1}$$

Also over $\mathbb{Z}_m$, the only affine relation which is true for *all program states* is the relation $\mathbf{0} = (0, \ldots, 0)^{\mathrm{t}}$. Since the initial state is arbitrary, an affine relation a is thus valid at a program point u iff $W_r\, a = \mathbf{0}$ for all runs r that reach u.

This is good news, since it shows that as in the case of fields, we may use the set $\mathcal{W} = \{W_r \mid r \text{ reaches } u\}$ to solve the validity problem for affine relations by setting up and solving the linear equation system $Wa = \mathbf{0}$, $W \in \mathcal{W}$. While in our case this set is finite because $\mathbb{Z}_m$ is finite, it can be large. We are thus left with the problem of representing $\mathcal{W}$ compactly. In the case of fields, we could observe that the set of $(k+1) \times (k+1)$ matrices forms a vector space. Over $\mathbb{Z}_m$ this is no longer the case. However, this set is still a $\mathbb{Z}_m$-*module* isomorphic to $\mathbb{Z}_m^{(k+1)^2}$. We observe that as in the case of fields we can use a generating system of the submodule $\langle\mathcal{W}\rangle$ instead of $\mathcal{W}$ to set up this linear equation system without changing the set of solutions. By Theorem 1, $\langle\mathcal{W}\rangle$ can be described by a generating system of at most $(k+1)^2$ matrices. Therefore, in order to determine the set of all affine relations at program point u, it suffices to compute a set of generators for the module $\langle\{W_r \mid r \text{ reaches } u\}\rangle$. This is the contents of the next theorem:

Theorem 3. *Assume we are given a generating system G of cardinality at most $(k+1)^2$ for the set $\langle\{W_r \mid r \text{ reaches } u\}\rangle$. Then we have:*

a) The affine relation $a \in \mathbb{Z}_m^{k+1}$ is valid at u iff $W\,a = \mathbf{0}$ for all $W \in G$.
b) A generating system for the $\mathbb{Z}_m$-submodule of all affine relations valid at program point u can be computed in time $\mathcal{O}(k^4 \cdot (k + \log(w)))$.

Proof. We only consider the complexity estimation. By statement a), the affine relation a is valid at u iff a is a solution of all the equations

$$\textstyle\sum_{j=0}^{k} w_{ij}\mathbf{a}_j \;=\; 0$$

for each matrix $W = (w_{ij}) \in G$ and $i = 0, \ldots, k$. The generating system G contains at most $(k+1)^2$ matrices each of which contributes $k+1$ equations. First, we can bring this set into triangular form. By Theorem 1, this can be done in time $\mathcal{O}(k^4 \cdot (k + \log(w)))$. The resulting system has at most $k+1$ equations. By Theorem 2, a generating system for the $\mathbb{Z}_m$-module of solutions of this system can be computed with $\mathcal{O}(\log(w) \cdot k^3)$ operations. The latter amount, however, is dominated by the former, and the complexity statement follows. □

As in the case of fields, we are left with the task of computing, for every program point u, a generating system for $\langle\{W_r \mid r \text{ reaches } u\}\rangle$. Following the approach in [11], we compute this submodule of $\mathbb{Z}_m^{(k+1)^2}$ as an abstract interpretation of a constraint system for set of runs reaching u. From Section 2 we know that $\mathbf{Sub}(\mathbb{Z}_m^{(k+1)^2})$ is a complete lattice of height $\mathcal{O}(k^2 \cdot w)$ such that we can compute fixpoints effectively. The desired abstraction of run sets is given by $\alpha : 2^{\mathsf{Runs}} \to \mathbf{Sub}(\mathbb{Z}_m^{(k+1)^2})$, $\alpha(R) = \langle\{W_r \mid r \in R\}\rangle$. Indeed, the mapping α is monotonic (w.r.t. subset ordering on sets of runs and submodules) and commutes with arbitrary unions. Similar to the case of fields, we set up a constraint system. The variables in the new constraint system take submodules of $\mathbb{Z}_m^{(k+1)^2}$ as values:

[S$_\alpha$1]	$\mathbf{S}_\alpha(q)$	$\supseteq \mathbf{S}_\alpha(\mathsf{ret}_q)$	
[S$_\alpha$2]	$\mathbf{S}_\alpha(\mathsf{st}_q)$	$\supseteq \langle\{I_{k+1}\}\rangle$	
[S$_\alpha$3]	$\mathbf{S}_\alpha(v)$	$\supseteq \mathbf{S}_\alpha(u) \circ \langle\{W_{\mathbf{x}_j:=t}\}\rangle$	for an edge (u,v) labeled $\mathbf{x}_j := t$
[S$_\alpha$4]	$\mathbf{S}_\alpha(v)$	$\supseteq \mathbf{S}_\alpha(u) \circ \langle\{W_{\mathbf{x}_j:=0}, W_{\mathbf{x}_j:=1}\}\rangle$	for an edge (u,v) labeled $\mathbf{x}_j := ?$
[S$_\alpha$5]	$\mathbf{S}_\alpha(v)$	$\supseteq \mathbf{S}_\alpha(u) \circ \mathbf{S}_\alpha(q)$	for an edge (u,v) calling q
[R$_\alpha$1]	$\mathbf{R}_\alpha(\mathbf{Main})$	$\supseteq \langle\{I_{k+1}\}\rangle$	
[R$_\alpha$2]	$\mathbf{R}_\alpha(q)$	$\supseteq \mathbf{R}_\alpha(u)$	for an edge $(u,\_)$ calling q
[R$_\alpha$3]	$\mathbf{R}_\alpha(u)$	$\supseteq \mathbf{R}_\alpha(q) \circ \mathbf{S}_\alpha(u)$	if u is a program point of q

The variable $\mathbf{S}_\alpha(q)$ is meant to capture the abstract effect of the procedure q. By the constraints $\mathbf{S}_\alpha 1$, this value is obtained as the module of transformations $\mathbf{S}_\alpha(\mathsf{ret}_q)$ accumulated for the return point ret_q of q. According to $\mathbf{S}_\alpha 2$, this accumulation starts at the start point st_q with (the module generated by) the identity transformation. The constraints $\mathbf{S}_\alpha 3$ and $\mathbf{S}_\alpha 4$ deal with affine and nondeterministic assignments, respectively, while the constraints $\mathbf{S}_\alpha 5$ correspond to calls. The abstract effects of procedures are then used in the second part of the

constraint system to determine for every procedure and program point the module of transformations induced by reaching runs. The constraint $\mathbf{R}_\alpha 1$ indicates that we start before the call to **Main** with the identity transformation. The constraints $\mathbf{R}_\alpha 2$ indicate that transformations reaching a procedure should comprise all transformation reaching its calls. Finally, the constraints $\mathbf{R}_\alpha 3$ state that the transformations for a program point u of some procedure q should contain the composition of transformations reaching q with the transformation accumulated for u from the start point st_q of q.

In this constraint system, the operator "$\circ$" abstracts concatenation of run sets. As in the case of fields, it is defined in terms of matrix multiplication by

$$M_1 \circ M_2 = \langle \{A_1 A_2 \mid A_i \in M_i\} \rangle$$

for sets of matrices $M_1, M_2 \subseteq \mathbb{Z}_m^{(k+1)^2}$. An edge annotated by $\mathbf{x}_j := ?$ induces the set of all runs $\mathbf{x}_j := c$, $c \in \mathbb{Z}_m$. As in the case of fields, however, the module spanned by the matrices $W_{\mathbf{x}_j:=c}, c \in \mathbb{Z}_m$, is generated by the two matrices $W_{\mathbf{x}_j:=0}$ and $W_{\mathbf{x}_j:=1}$. Therefore, these two suffice to abstract the effect of $\mathbf{x}_j := ?$.

Again as in the case of fields, the constraint system from above can be solved by computing on generating systems. In contrast to fields, however, we no longer have the notion of a basis available. Instead, we rely on sets of generators in triangular form. In order to avoid the need to solve a system of equations over $\mathbb{Z}_m$ fully whenever a new vector is added to a set of generators, we use our algorithm from Theorem 1 to bring the enlarged set again into triangular form. A set of generators, thus, may have to be changed — even if the newly added vector is implied by the original one. The update, however, then *decreases* the rank of the set of generators implying that ultimately stabilization is detected.

We assume that the basic statements in the given program have size $\mathcal{O}(1)$. Thus, we measure the size n of the given program by $|N| + |E|$. We obtain:

Theorem 4. *For every program of size n with k variables the following holds:*

a) The values: $\langle \{W_r \mid r \text{ reaches } X\} \rangle$, X a procedure or program point, equal the components of the least solution of the constraint system for the $\mathbf{R}_\alpha(X)$.
b) These values can be computed in time $\mathcal{O}(w \cdot n \cdot k^6 \cdot (k^2 + \log(w)))$.
c) The sets of all valid affine relations at program point u, $u \in N$, can be computed in time $\mathcal{O}(w \cdot n \cdot k^6 \cdot (k^2 + \log(w)))$. □

A full proof of Theorem 4 can be found in [10]. In our main application, w equals 32 or 64. The term $\log(w)$ in the complexity estimation accounts for the necessary rank computations. In our application, $\log(w)$ equals 5 or 6 and thus is easily dominated by k^2. We conclude that the extra overhead over the corresponding complexity from [11] for the analysis over fields (w.r.t. unit cost for basic arithmetic operations) essentially consists in one extra factor $w = 32$ or 64 which is due to the increased height of the lattice used. We expect, however, that fixpoint computations in practice will not exploit full maximal lengths of ascending chains in the lattice but stabilize much earlier.

Example 5. Consider the inter-procedural program from Figure 2 and assume that we want to infer all valid affine relations modulo $\mathbb{Z}_m$ for $m = 2^{32}$, and let

c abbreviate 7654321. The weakest precondition transformers for $s_1 \equiv \mathbf{x}_1 := 7654321 \cdot \mathbf{x}_1; \mathbf{x}_2 := \mathbf{x}_1 + \mathbf{x}_2$ and $s_2 \equiv \mathbf{x}_1 := 69246289 \cdot \mathbf{x}_1; \mathbf{x}_2 := \mathbf{x}_1 + \mathbf{x}_2$ are:

$$B_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & c & c \\ 0 & 0 & 1 \end{pmatrix} \qquad B_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & c^{-1} & c^{-1} \\ 0 & 0 & 1 \end{pmatrix}$$

since $c \cdot 69246289 = 1 \bmod 2^{32}$. For $\mathbf{R}_\alpha(q)$, we find the matrices I_{k+1} and

$$P_1 = B_1 \cdot B_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & c+1 \\ 0 & 0 & 1 \end{pmatrix}$$

None of these is subsumed by the other where the corresponding triangular set of generators is given by $G_1 = \{I_{k+1}, P\}$ where

$$P = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & c+1 \\ 0 & 0 & 0 \end{pmatrix}$$

The next iteration then results in the matrix

$$P_2 = B_1 \cdot P_1 \cdot B_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & (c+1)^2 \\ 0 & 0 & 1 \end{pmatrix}$$

Since $P_2 = I_{k+1} + (c+1) \cdot P$, computing a triangular set G_2 of generators for G_1 together with P_2 will result in $G_2 = G_1$, and the fixpoint iteration terminates.

In order to obtain the weakest precondition transformers, e.g., for the end-point 6 of **Main**, we additionally need the transformation B_0 for $\mathbf{x}_1 := 5; \mathbf{x}_2 := 0$:

$$B_0 = \begin{pmatrix} 1 & 5 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Using the set $\{I_{k+1}, P\}$ of generators for $\mathbf{S}_\alpha(q)$, we thus obtain for $\mathbf{R}_\alpha(6)$ the generators:

$$W_1 = B_0 \cdot I_{k+1} = \begin{pmatrix} 1 & 5 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad W_2 = B_0 \cdot P = \begin{pmatrix} 0 & 0 & 5c+5 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

This gives us the following equations for the affine relations at the exit of **Main**:

$$\begin{aligned} \mathbf{a}_0 + 5\mathbf{a}_1 \qquad\qquad &= 0 \\ (5c+5)\mathbf{a}_2 &= 0 \end{aligned}$$

Solving this equation system over $\mathbb{Z}_m$ according to Theorem 2 shows that the set of all solutions is generated by:

$$a = \begin{pmatrix} -5 \\ 1 \\ 0 \end{pmatrix} \qquad a' = \begin{pmatrix} 0 \\ 0 \\ 2^{31} \end{pmatrix}$$

The vector a means $-5 + \mathbf{x}_1 = 0$ or, equivalently, $\mathbf{x}_1 = 5$. The vector a' means that $2^{31} \cdot \mathbf{x}_2 = 0$ or, equivalently, $\mathbf{x}_2$ is *even*. Both relations are non-trivial and could not have been derived by using the corresponding analysis over $\mathbb{Q}$. □

The runtime of our inter-procedural analysis is linear in the program size n but polynomial in the number of program variables k of a rather high degree. In [8], we have presented an efficient algorithm which in absence of procedures, computes all valid affine relations in time $\mathcal{O}(n \cdot k^3)$ — given that all arithmetic operations count for 1. This algorithm improves on the original algorithm by Karr [6] for the same problem by one factor of k. Due to lack of space we will neither rephraze this nor Karr's original algorithm but remark that both algorithms assume that the program variables take values in a field, namely $\mathbb{Q}$ — but any other field $\mathbb{Z}_p$ (p prime) would do as well. Similar to the algorithm from [11], they compute with finite-dimensional vector spaces represented through sets of generators. Just as in our exposition for the interprocedural analysis, we obtain a sound and complete *intraprocedural* analysis if we replace the vector spaces of the algorithm in [8] with the corresponding $\mathbb{Z}_m$-modules and use the algorithms from Section 2 for reducing the cardinalities of sets of generators and solving sets of homogeneous equations. Summarizing, we obtain:

Theorem 5. *Consider an affine program of size n with k variables but without procedures. Then for $m = 2^w$, the set of* all *affine relations at all program points which are valid over $\mathbb{Z}_m$ can be computed in time $\mathcal{O}(w \cdot n \cdot k^2 \cdot (k + \log(w)))$.* □

4 Conclusion

We have presented sound and complete inter- and intraprocedural algorithms for computing valid affine relations in affine programs over rings $\mathbb{Z}_m$ where $m = 2^w$. These techniques allow us to analyze integer arithmetic in programming languages like Java precisely (upto abstraction of guards). Our new algorithms were obtained from the corresponding algorithms in [11] and [8] by replacing techniques for vector spaces with techniques for $\mathbb{Z}_m$-modules. The difficulty here is that for $w > 1$, the ring $\mathbb{Z}_m$ has zero divisors — implying that not every element in the ring is invertible. Since we maintained the top-level structure of the analysis algorithms, we achieved the same complexity bounds as in the case of fields — upto one extra factor w due to the increased height of the used complete lattices. We carefully avoid explicit computation of inverses in our algorithms for reducing sets of generators and for solving homogeneous linear equation systems. Otherwise the complexity estimates for the algorithms would be worse because computation of inverses cannot reasonably be assumed constant time.

Our algorithms have the clear advantage that their arithmetic operations can completely be performed within the ring $\mathbb{Z}_m$ of the target language to be analyzed. All problems with excessively long numbers are thus resolved. In [10] we also show how to extend the analyzes to $\mathbb{Z}_m$ for an arbitrary $m > 2$.

We remark that in [11], we also have shown how the linear algebra methods over fields can be extended to deal with local variables and return values of procedures besides just global variables. These techniques immediately carry over to arithmetic in $\mathbb{Z}_m$. The same is true for the generalization to the inference of all valid *polynomial* relations up to a fixed degree bound.

One method to deal with *inequalities* instead of equalities is to use *polyhedra* for abstracting sets of vectors [1]. It is a challenging question what kind of impact modular arithmetic has on this abstraction.

Acknowledgments. We thank Martin Hofmann for pointing us to the topic of analyzing modular arithmetic and the anonymous referees for valuable comments.

References

1. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *5th ACM Symp. on Principles of Programming Languages (POPL)*, pages 84–97, 1978.
2. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
3. P. Granger. Static Analysis of Linear Congruence Equalities among Variables of a Program. In *Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT)*, pages 169–192. LNCS 493, Springer-Verlag, 1991.
4. S. Gulwani and G. Necula. Discovering Affine Equalities Using Random Interpretation. In *30th ACM Symp. on Principles of Programming Languages (POPL)*, pages 74–84, 2003.
5. J. Hafner and K. McCurley. Asymptotically Fast Triangularization of Matrices over Rings. *SIAM J. of Computing*, 20(6):1068–1083, 1991.
6. M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.
7. J. Leroux. *Algorithmique de la Vérification des Systèmes à Compteurs: Approximation et Accélération*. PhD thesis, Ecole Normale Supérieure de Cachan, 2003.
8. M. Müller-Olm and H. Seidl. A Note on Karr's Algorithm. In *31st Int. Coll. on Automata, Languages and Programming (ICALP)*, pages 1016–1028. Springer Verlag, LNCS 3142, 2004.
9. M. Müller-Olm and H. Seidl. Computing Polynomial Program Invariants. *Information Processing Letters (IPL)*, 91(5):233–244, 2004.
10. M. Müller-Olm and H. Seidl. Interprocedural Analysis of Modular Arithmetic. Technical Report 789, Fachbereich Informatik, Universität Dortmund, 2004.
11. M. Müller-Olm and H. Seidl. Precise Interprocedural Analysis through Linear Algebra. In *31st ACM Symp. on Principles of Programming Languages (POPL)*, pages 330–341, 2004.
12. T. Reps, S. Schwoon, and S. Jha. Weighted Pushdown Systems and their Application to Interprocedural Dataflow Analysis. In *Int. Static Analysis Symposium (SAS)*, pages 189–213. LNCS 2694, Springer-Verlag, 2003.
13. A. Storjohann. *Algorithms for Matrix Canonical Forms*. PhD thesis, ETH Zürich, Diss. ETH No. 13922, 2000.

Forward Slicing by Conjunctive Partial Deduction and Argument Filtering*

Michael Leuschel[1] and Germán Vidal[2]

[1] School of Electronics and Computer Science,
University of Southampton & Institut für Informatik,
Heinrich-Heine Universität Düsseldorf
mal@ecs.soton.ac.uk
[2] DSIC, Technical University of Valencia,
Camino de Vera S/N, E-46022 Valencia, Spain
gvidal@dsic.upv.es

Abstract. Program slicing is a well-known methodology that aims at identifying the program statements that (potentially) affect the values computed at some point of interest. Within imperative programming, this technique has been successfully applied to debugging, specialization, reuse, maintenance, etc. Due to its declarative nature, adapting the slicing notions and techniques to a logic programming setting is not an easy task. In this work, we define the first, semantics-preserving, forward slicing technique for logic programs. Our approach relies on the application of a conjunctive partial deduction algorithm for a precise propagation of information between calls. We do not distinguish between static and dynamic slicing since partial deduction can naturally deal with both static and dynamic data. A slicing tool has been implemented in ECCE, where a post-processing transformation to remove redundant arguments has been added. Experiments conducted on a wide variety of programs are encouraging and demonstrate the usefulness of our approach, both as a classical slicing method and as a technique for code size reduction.

1 Introduction

Program slicing is a fundamental operation that has been successfully applied to solve many software engineering tasks, like, e.g., program understanding, maintenance, specialization, debugging, reuse, etc. Slicing was originally introduced by Weiser [32]—in the context of imperative programs—as a debugging technique. Despite its potential applications, we found very few approaches to slicing in logic programming (some notable exceptions are, e.g., [10, 27, 28, 30, 33]).

Informally, a *program slice* consists of those program statements which are (potentially) related with the values computed at some program point and/or variable, referred to as a *slicing criterion*. Program slices are usually computed

* This work was partially funded by the IST programme of the European Commission, Future and Emerging Technologies under the IST-2001-38059 ASAP project and by the Spanish *Ministerio de Educación y Ciencia* (ref. TIN2004-00231).

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 61–76, 2005.

from a *program dependence graph* [5] that makes explicit both the data and control dependences for each operation in a program. Program dependences can be traversed backwards and forwards—from the slicing criterion—giving rise to so-called *backward* and *forward* slicing. Additionally, slices can be *static* or *dynamic*, depending on whether a concrete program's input is provided or not. More detailed information on program slicing can be found in [12, 29].

Recently, Vidal [31] introduced a novel approach to forward slicing of lazy functional logic programs. This work exploits the similarities between slicing and partial evaluation—already noticed in [25]—to compute forward slices by a slight modification of an existing partial evaluation scheme [2]. The main requirement of [31] is that the underlying partial evaluation algorithm should be—in the terminology of [26]—both *monovariant* and *monogenetic* in order to preserve the structure of the original program. Unfortunately, this requirement also restricts the precision of the computed slices.

In this work, we extend the approach of [31] in several ways. First, we adapt it to the logic programming setting. Second, we consider a polyvariant and polygenetic partial evaluation scheme: the *conjunctive* partial deduction algorithm of [3] with control based on characteristic trees [9, 18, 19]. Therefore, the computed slices are significantly more precise than those of the previous approach. Furthermore, since the basic partial deduction algorithm is kept unmodified, it can easily be implemented on top of an existing partial deduction system (in our case, ECCE [19]). Finally, we use the redundant argument filtering transformation of [21] to slice out unnecessary arguments of predicates (in addition to slicing out entire clauses).

The combination of these two approaches, [31] and [21], together with a special-purpose slicing code generator, gives rise to a simple but powerful forward slicing technique. We also pay special attention to using slicing for code size reduction. Indeed, within the ASAP project [1], we are looking at resource-aware specialization techniques, with the aim of adapting software for pervasive devices with limited resources. We hence also analyze to what extent our approach can be used as an effective code size reduction technique, to reduce the memory footprint of a program.

Our main contributions are the following. We introduce the first, semantics-preserving, forward slicing technique for logic programs that produces executable slices. While traditional approaches in the literature demand different techniques to deal with static and dynamic slicing, our scheme is general enough to produce both static and dynamic slices. In contrast to [31], the restriction to adopt a monovariant/monogenetic partial evaluation algorithm is not needed. Dropping this restriction is important as it allows us to use more powerful specialization schemes and, moreover, we do not need to modify the basic algorithm, thus easing the implementation of a slicing tool (i.e., only the code generation phase should be changed). We illustrate the usefulness of our approach on a series of benchmarks, and analyze its potential as a code-size reduction technique.

The paper is organized as follows. After introducing some foundations in the next section, Sect. 3 presents our basic approach to the computation of

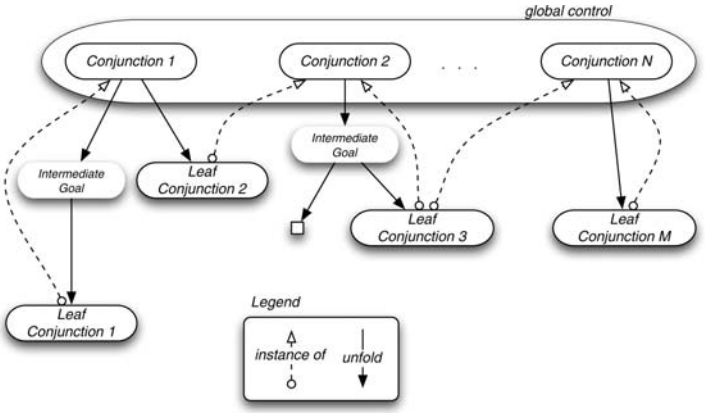


Fig. 1. The Essence of Conjunctive Partial Deduction

forward slices. Then, Sect. 4 considers the inclusion of a post-processing phase for argument filtering. Section 5 illustrates our technique by means of a detailed example, while Sect. 6 presents an extensive set of benchmarks. Finally, Sect. 7 compares some related works and concludes. More details and missing proofs can be found in [22].

2 Background

Partial evaluation [13] has been applied to many programming languages, including functional, imperative, object-oriented, logic, and functional logic programming languages. It aims at improving the overall performance of programs by pre-evaluating parts of the program that depend solely on the *static* input.

In the context of logic programming, full input to a program P consists of a goal G and evaluation corresponds to constructing a complete SLDNF-tree for $P \cup \{G\}$. For partial evaluation, the static input takes the form of a goal G' which is more general (i.e., less instantiated) than a typical goal G at runtime. In contrast to other programming languages, one can still execute P for G' and (try to) construct an SLDNF-tree for $P \cup \{G'\}$. However, since G' is not yet fully instantiated, the SLDNF-tree for $P \cup \{G'\}$ is usually infinite and ordinary evaluation will not terminate. A technique which solves this problem is known under the name of *partial deduction* [23]. Its general idea is to construct a finite number of finite, but possibly incomplete[1] SLDNF-trees and to extract from these trees a new program that allows any instance of the goal G' to be executed.

Conjunctive partial deduction (CPD) [3] is an extension of partial deduction that can achieve effects such as *deforestation* and *tupling* [24]. The essence of CPD can be seen in Fig. 1. The so-called global control of CPD generates a set $C = \{C_1, \ldots, C_n\}$ of *conjunctions* whereas the local control generates for each

[1] An SLDNF-tree is *incomplete* if, in addition to success and failure leaves, it also contains leaves where no literal has been selected for a further derivation step.

conjunction a possibly incomplete SLDNF-tree τ_i (a process called *unfolding*). The overall goal is to ensure that every leaf conjunction is either an instance of some C_i or can be split up into sub-conjunctions, each of which is an instance of some conjunction in C. This is called the *closedness* condition, and guarantees correctness of the specialized program which is then extracted by:

- generating one specialized predicate per conjunction in C (and inventing a new predicate name for it), and by producing
- one specialized clause—a *resultant*—per non-failing branch of τ_i.

A single resolution step with a specialized clause now corresponds to performing *all* the resolutions steps (using original program clauses) on the associated branch. Closedness can be ensured by various algorithms [16]. Usually, one starts off with an initial conjunction, unfolds it using some "*unfolding rule*" (a function mapping a program P and a goal G to an SLDNF-tree for $P \cup \{G\}$) and then adds all uncovered[2] leaf conjunctions to C, in turn unfolding them, and so forth. As this process is usually non-terminating, various "*generalization*" operations are applied, which, for example, can replace several conjunctions in C by a single less instantiated one. One useful foundation for the global control is based on so-called *characteristic trees*, used for example by the SP [7] and ECCE [19] specialization systems. We describe them in more detail below, as they turn out to be important for slicing.

Characteristic trees were introduced in partial deduction in order to capture all the relevant aspects of specialization. The following definitions are taken from [19] (which in turn were derived from [9] and the SP system [7]).

Definition 1 (characteristic path). *Let G_0 be a goal, and let P be a normal program whose clauses are numbered. Let $G_0, \ldots, G_n$ be the goals of a finite, possibly incomplete SLDNF-derivation D of $P \cup \{G_0\}$. The characteristic path of the derivation D is the sequence $\langle l_0 : c_0, \ldots, l_{n-1} : c_{n-1} \rangle$, where l_i is the position of the selected literal in G_i, and c_i is defined as follows:*

- *if the selected literal is an atom, then c_i is the number of the clause chosen to resolve with G_i;*
- *if the selected literal is $\neg p(\bar{t})$, then c_i is the predicate p.*

Note that an SLDNF-derivation D can be either failed, incomplete, successful, or infinite. As we will see below, characteristic paths will only be used to characterize *finite* and *nonfailing* derivations. Once the top-level goal is known, the characteristic path is sufficient to reconstruct all the intermediate goals as well as the final one.

Now that we have characterized derivations, we can characterize goals through the derivations in their associated SLDNF-trees.

Definition 2 (characteristic tree). *Let G be a goal, P a normal program, and τ a finite SLDNF-tree for $P \cup \{G\}$. Then the characteristic tree $\hat{\tau}$ of τ is*

[2] I.e., those conjunctions which are not an instance of a conjunction in C.

the set containing the characteristic paths of the nonfailing SLDNF-derivations associated with the branches of τ. $\hat{\tau}$ *is called a characteristic tree if and only if it is the characteristic tree of some finite SLDNF-tree.*

Let U *be an unfolding rule such that* $U(P, G) = \tau$. *Then* $\hat{\tau}$ *is also called the characteristic tree of* G *(in* P*) via* U. *We introduce the notation* $chtree(G, P, U) = \hat{\tau}$. *We also say that* $\hat{\tau}$ *is a characteristic tree of* G *(in* P*) if it is* the *characteristic tree of* G *(in* P*) via some unfolding rule* U.

When characteristic trees are used to control CPD, the basic algorithm returns a set of characteristic conjunctions, $\tilde{C}$, that fulfills the conditions for the correctness of the specialization process. A *characteristic conjunction* is a pair $(C, \hat{\tau})$, where C is a conjunction of literals—a goal—and $\hat{\tau} = chtree(C, P, U)$ is a characteristic tree for some program P and unfolding rule U. From this set of characteristic conjunctions, the specialized program is basically obtained by unfolding and renaming.

3 Extracting Executable Forward Slices

Within imperative programming, the definition of a slicing criterion depends on whether one considers static or dynamic slicing. In the former case, a slicing criterion is traditionally defined as a pair (p, v) where p is a program statement and v is a subset of the program's variables. Then, a forward slice consists of those statements which are dependent on the slicing criterion (i.e., on the values of the variables v that appear in p), a statement being *dependent* on the slicing criterion if the values computed at that statement depend on the values computed at the slicing criterion or if the values computed at the slicing criterion determine if the statement under consideration is executed [29]. As for dynamic slicing, a slicing criterion is often defined as a triple (d, i, v), where d is the input data for the program, i denotes the i-th element of the execution history, and v is a subset of the program's variables.

Adapting these notions to the setting of logic programming is not immediate. There are mainly two aspects that one should take into account:

- The execution of partially instantiated goals—thanks to the use of logic variables—makes it unclear the distinction between static and dynamic slicing.
- The lack of explicit control flow, together with the absence of side effects, makes unnecessary to consider a particular trace of the program's execution for dynamic slicing.

Therefore, we define a *slicing criterion* simply as a goal[3]. Typically, the goal will appear in the code of the source program. However, we lift this requirement for simplicity since it does affect to the forthcoming developments. A forward slice should thus contain a *subset* of the original program with those clauses that are reachable from the slicing criterion. Similarly to [27], the notion of "subset" is

[3] If we fix an entry point to the program and restrict ourselves to a particular evaluation strategy (as in Prolog), one can still consider a concrete trace of the program. In this case, however, a standard tracer would suffice to identify the interesting goal.

formalized in terms of an abstraction relation, to allow arguments to be removed, or rather replaced by a special term:

Definition 3 (term abstraction). *Let $\top_t$ be the empty term (i.e., an unnamed existentially quantified variable, like the anonymous variable of Prolog). A term t is an abstraction of term t', in symbols $t \succeq t'$, iff $t = \top_t$ or $t = t'$.*

Definition 4 (literal abstraction). *An atom $p(t_1, \ldots, t_n)$ is an abstraction of atom $q(t'_1, \ldots, t'_m)$, in symbols $p(t_1, \ldots, t_n) \succeq q(t'_1, \ldots, t'_m)$, iff $p = q$, $n = m$, and $t_i \succeq t'_i$ for all $i = 1, \ldots, n$. A negative literal $\neg P$ is an abstraction of a negative literal $\neg Q$ iff $P \succeq Q$.*

Definition 5 (clause abstraction). *A clause c is an abstraction of a clause $c' = L'_0 \leftarrow L'_1, \ldots, L'_n$, in symbols $c \succeq c'$, iff $c = L_0 \leftarrow L_1, \ldots, L_n$ and $L_i \succeq L'_i$ for all $i \in \{1, \ldots, n\}$.*

Definition 6 (program abstraction). *A normal program[4] $P = (c_1, \ldots, c_n)$ is an abstraction of normal program $P' = (c'_0, \ldots, c'_m)$, in symbols $P \succeq P'$, iff $n \leq m$ and there exists a subsequence $(s_1, \ldots, s_n)$ of $(1, \ldots, m)$ such that $c_i \succeq c'_{s_i}$ for all $i \in \{1, \ldots, n\}$.*

Informally, a program P is an abstraction of program P' if it can be obtained from P' by clause deletion and by replacing some predicate arguments by the empty term $\top_t$. In the following, P is a *slice* of program P' iff $P \succeq P'$. Trivially, program slices are normal programs.

Definition 7 (correct slice). *Let P be a program and G a slicing criterion. A program P' is a correct slice of P w.r.t. G iff P' is a slice of P (i.e., $P' \succeq P$) and the following conditions hold:*

- *$P \cup \{G\}$ has an SLDNF-refutation with computed answer θ if and only if $P' \cup \{G\}$ does, and*
- *$P \cup \{G\}$ has a finitely failed SLDNF-tree if and only if $P' \cup \{G\}$ does.*

Traditional approaches to program slicing rely on the construction of some data structure which reflects the data and control dependences in a program (like, e.g., the *program dependence graphs* of [5]). The key contribution of this paper is to show that CPD can actually play such a role.

Roughly speaking, our slicing technique proceeds as follows. Firstly, given a program P and a goal G, a CPD algorithm based on characteristic trees is applied. The use of characteristic trees is relevant in our context since they record the clauses used during the unfolding of each conjunction. The complete algorithm outputs a so-called *global tree*—where each node is a characteristic conjunction—which represents an abstraction of the execution of the considered goal. In fact, this global tree contains information which is similar to that in a

[4] We consider that programs are *sequences* of clauses in order to enforce the preservation of the syntax of the original program.

program dependence graph (e.g., dependences among predicate calls). In standard conjuntive partial deduction, the characteristic conjunctions, $\tilde{C}$, in the computed global tree are unfolded—following the associated characteristic trees—to produce a correct specialization of the original program (after renaming). In order to compute a forward slice, only the code generation phase of the CPD algorithm should be changed: now, we use the characteristic tree of each conjunction in $\tilde{C}$ to determine which clauses of the original program have been used and, thus, should appear in the slice.

Given a characteristic path δ, we define $cl(\delta)$ as the set of clause numbers in this path, i.e., $cl(\delta) = \{c \mid \langle l : c\rangle$ appears in δ and c is a clause number$\}$. Program slices are then obtained from a set of characteristic trees as follows:

Definition 8 (forward slicing). *Let P be a normal program and G be a slicing criterion. Let $\tilde{C}$ be the output of the CPD algorithm (a set of characteristic conjunctions) and T be the characteristic trees in $\tilde{C}$. A forward slice of P w.r.t. G, denoted by $slice_T(P)$, contains those clauses of P that appear in some characteristic path of T. Formally, $slice_T(P) = \cup_{\hat{\tau}\in T} \{cl(\delta) \mid \delta \in \hat{\tau}\}$.*

The correctness of the forward slicing method is stated as follows:

Theorem 1. *Let P be a normal program and G be a slicing criterion. Let P' be a forward slice according to Def. 8. Then, P' is a correct slice of P w.r.t. G.*

The proof can be found in [22]. Our slicing technique produces *correct* forward slices and, moreover, is more flexible than previous approaches in the literature. In particular, in can be used to perform both dynamic and static forward slicing with a modest implementation effort, since only the code generation phase of the CPD algorithm should be changed.

4 Improving Forward Slices by Argument Filtering

The method of Def. 8 has been fully implemented in ECCE, an off-the-shelf partial evaluator for logic programs based on CPD and characteristic trees. In practice, however, we found that computed slices often contain redundant arguments that are not relevant for the execution of the slicing criterion. In order to further refine the computed slices and be able to slice out unnecessary arguments of predicates, we use the redundant argument filtering transformations (RAF) of [21].

RAF is a technique which detects certain redundant arguments (finding all redundant arguments is undecidable in general [21]). Basically, it detects those arguments which are existential and which can thus be safely removed. RAF is very useful when performed after CPD. Redundant arguments also arise when one re-uses generic predicates for more specific purposes. For instance, let us define a `member/2` predicate by re-using a generic `delete/3` predicate:

```
member(X,L) :- delete(X,L,DL).
delete(X,[X|T],T).     delete(X,[Y|T],[Y|DT]) :- delete(X,T,DT).
```

Here, the third argument of `delete` is redundant and will be removed by the partial evaluator ECCE if RAF is enabled:

```
member(X,L) :- delete(X,L).
delete(X,[X|T]).        delete(X,[Y|T]) :- delete(X,T).
```

The ECCE system also contains the reverse argument filtering (FAR) of [21] ("reverse" because the safety conditions are reversed w.r.t. RAF). While RAF detects existential arguments (which might return a computed answer binding), FAR detects arguments which can be non-existential and non-ground but whose value is never used (and for which no computed answer binding will be returned). Consider, e.g., the following program:

```
p(X) :-  q(f(X)).      q(Z).
```

Here, the argument of `q(f(X))` is not a variable but the value is never used. The ECCE system will remove this argument if FAR is enabled:

```
p(X) :-  q.      q.
```

The elimination of redundant arguments turns out to be quite useful to remove unnecessary arguments from program slices (see next section). Only one extension is necessary in our context: while redundant arguments are deleted in [21], we replace them by the special symbol $\top_t$ so that the filtered program is still a slice—an abstraction—of the original program. The correctness of the extended slicing algorithm then follows from Theorem 1 and the results in [21].

5 Forward Slicing in Practice

In this section, we illustrate our approach to the computation of forward slices through some selected examples. Consider the program in Fig. 2 which defines an interpreter for a simple language with constants, variables, and some predefined functions. First, we consider the following slicing criterion:

```
slice1(X) :- int(minus(cst(4),plus(fun(one),cst(2))),[xx],[11],X).
```

The slice computed by ECCE w.r.t. this slicing criterion is as follows:

```
int(cst(X),_,_,X).

int(plus(X,Y),Vars,Vals,Res) :-
   int(X,Vars,Vals,RX), int(Y,Vars,Vals,RY), Res is RX+RY.
int(minus(X,Y),Vars,Vals,Res) :-
   int(X,Vars,Vals,RX), int(Y,Vars,Vals,RY), Res is RX-RY.
int(fun(X),Vars,Vals,Res) :- def0(X,Def), int(Def,Vars,Vals,Res).

def0(one,cst(1)).

slice1(X) :- int(minus(cst(4),plus(fun(one),cst(2))),[xx],[11],X).
```

```
int(cst(X),_,_,X).
int(var(X),Vars,Vals,R) :- lookup(X,Vars,Vals,R).
int(plus(X,Y),Vars,Vals,Res) :-
    int(X,Vars,Vals,RX), int(Y,Vars,Vals,RY), Res is RX+RY.
int(minus(X,Y),Vars,Vals,Res) :-
    int(X,Vars,Vals,RX), int(Y,Vars,Vals,RY), Res is RX-RY.
int(fun(X),Vars,Vals,Res) :- def0(X,Def), int(Def,Vars,Vals,Res).
int(fun(X,Arg),Vars,Vals,Res) :-
    def1(X,Var,Def), int(Arg,Vars,Vals,ResArg),
    int(Def,[Var|Vars],[ResArg|Vals],Res).
def0(one,cst(1)).
def0(rec,fun(rec)).
def1(inc,xx,plus(var(xx),cst(1))).
def1(rec,xx,fun(rec,var(xx))).
lookup(X,[X|_],[Val|_],Val).
lookup(X,[Y|T],[_|ValT],Res) :- X \= Y, lookup(X,T,ValT,Res).
```

Fig. 2. A simple functional interpreter

Here, some predicates have been completely removed from the slice (e.g., `def1` or `lookup`), even though they are reachable in the predicate dependency graph. Furthermore, unused clauses are also removed, cutting down further the size of the slice. By applying the argument filtering post-processing, we get[5]

```
int(cst(X),*,*,X).

int(plus(X,Y),*,*,Res) :- int(X,*,*,RX), int(Y,*,*,RY), Res is RX+RY.
int(minus(X,Y),*,*,Res) :- int(X,*,*,RX), int(Y,*,*,RY), Res is RX-RY.
int(fun(X),*,*,Res) :- def0(X,Def), int(Def,*,*,Res).

def0(one,cst(1)).

slice1(X) :- int(minus(cst(4),plus(fun(one),cst(2))),*,*,X).
```

The resulting slice is executable and will produce the same result as the original program, e.g., the query `slice1(X)` returns the answer `X=1`. Note that this example could have been tackled by a *dynamic* slicing method, as a fully specified query was provided as the slicing criterion. It would be interesting to know how a dynamic slicer would compare against our technique, and whether we have lost any precision. In order to test this, we have implemented a simple dynamic slicer in SICStus Prolog using profiled code and extracting the used clauses using the `profile_data/4` built-in. The so extracted slice corresponds exactly to our result (without the argument filtering; see [22]), and hence no precision has been lost in this example.

[5] For clarity, in the examples we use "`*`" to denote the empty term $\top_t$. In practice, empty terms can be replaced by any term since they play no role in the computation.

In general, not only the code size of the slice is smaller but also the runtime can be improved. Thus, our forward slicing algorithm can be seen as a—rather conservative—partial evaluation method that guarantees that code size does not increase. For instance, it can be useful for resource aware specialization, when the (potential) code explosion of typical partial evaluators is unacceptable.

Our slicing tool can also be useful for program debugging. In particular, it can help the programmer to locate the source of an incorrect answer (or an unexpected loop; finite failure is preserved in Def. 7) since it identifies the clauses that could affect the computed answer, thus easing the correction of the program. Consider, e.g., that the definition of function `plus` contains a bug:

```
int(plus(X,Y),Vars,Vals,Res) :-
    int(X,Vars,Vals,RX), int(Y,Vars,Vals,RY), Res is RX-RY.
```

i.e., the programmer wrote `RX-RY` instead of `RX+RY`. Given the following goal:

```
slice2(X) :- int(plus(cst(1),cst(2)),[x],[1],X).
```

the execution returns the—incorrect—computed answer `X = -1`. By computing a forward slice w.r.t. `slice2(X)`, we get (after argument filtering) the following:

```
int(cst(X),*,*,X).
int(plus(X,Y),*,*,Res) :- int(X,*,*,RX), int(Y,*,*,RY), Res is RX-RY.
slice2(X) :- int(plus(cst(1),cst(2)),*,*,X).
```

This slice contains only 3 clauses and, thus, the user can easily detect that the definition of `plus` is wrong.

The previous two slices can be extracted by a dynamic slicing technique, since they do not involve a non-terminating goal. Now, we consider the following slicing criterion:

```
slice3(X) :- int(fun(rec),[aa,bb,cc,dd],[0,1,2,3],X).
```

Despite the fact that this goal has an infinite search space, our slicing tool returns the following slice (after argument filtering):

```
int(fun(X),*,*,*) :- def0(X,Def), int(Def,*,*,*).
def0(rec,fun(rec)).
slice3(X) :- int(fun(rec),*,*,*).
```

From this slice, the clauses which are responsible of the infinite computation can easily be identified.

6 Experimental Results

In this section, we show a summary of the experiments conducted on an extensive set of benchmarks. We used SICStus Prolog 3.11.1 (powerpc-darwin-7.2.0) and Ciao-Prolog 1.11 #221, running on a Powerbook G4, 1GHz, 1GByte of RAM.

Table 1. Speedups obtained by Specialization and by Slicing

Prolog System	SWI-Prolog		SICStus		Ciao	
Technique	Specialized	Sliced	Specialized	Sliced.	Specialized	Sliced
TOTAL	2.43	1.04	2.74	1.04	2.62	1.05
Average	5.23	1.07	6.27	1.09	11.26	1.09

The operating system was Mac OS 10.3. We also ran some experiments with SWI Prolog 5.2.0. The runtime was obtained by special purpose benchmarker files (generated automatically be ECCE) which execute the original and specialized programs without loop overhead. The code size was obtained by using the `fcompile` command of SICStus Prolog and then measuring the size of the compiled `*.ql` files. The total speedups were obtained by the formula $\frac{n}{\sum_{i=1}^{n} \frac{spec_i}{orig_i}}$ where n is the number of benchmarks, and $spec_i$ and $orig_i$ are the absolute execution times of the specialized/sliced and original programs respectively.[6] The total code size reduction was obtained by the formula $1 - \frac{\sum_{i=1}^{n} specsz_i}{\sum_{i=1}^{n} origsz_i}$ where n is the number of benchmarks, and $specsz_i$ and $origsz_i$ are the code sizes of the specialized/sliced and original programs respectively.

DPPD. We first compared the slicing tool with the default conjunctive specialization of ECCE on the DPPD library of specialization benchmarks [15]. In a sense these are not typical slicing scenarios, but nonetheless give an indication of the behavior of the slicing algorithm. The experiments also allow us to evaluate to what extent our technique is useful as an alternative way to specialize programs, especially for code size reduction. Finally, the use of the DPPD library allows comparison with other reference implementations (see, e.g., [3, 14] for comparisons with MIXTUS, SP and PADDY).

Table 1 (which is a summary of the full tables in [22]) shows the speedup of the ECCE default specialization and of our slicing algorithm. Timings for SWI Prolog, SICStus Prolog, and Ciao Prolog are shown. It can be seen that the average speedup of slicing is just 4%. This shows how efficient modern Prolog implementations are, and that little overhead has to be paid for adding extra clauses to a program. Anyway, the main purpose of slicing is not speedup, but reducing code size. In this case, slicing has managed an overall code size reduction of 26.2% whereas the standard specialization has increased the code size by 56%. In the worst case, the specialization has increased the code size by 493.5% (whereas slicing never increases the code size; see the full tables in [22]).

Slicing-Specific Benchmarks. Let us now turn our attention to four, more slicing-specific experiments. Table 2 contains the results of these experiments. The `inter_medium` benchmark is the simple interpreter of Sect. 5. The `ctl_trace`

[6] Observe that this is different from the average of the speedups (which has the disadvantage that big slowdowns are not penalized sufficiently).

Table 2. Slicing Specific Benchmarks

	Slicing Time	Runtime		Size		
Benchmark	ms	Original ms	Sliced speedup	Original Bytes	Sliced Bytes	Reduction %
inter_medium	20	117	1.06	4798	1578	67.1%
lambdaint	390	177	1.29	7389	4769	35.5%
ctl_trace	1940	427	1.35	8053	4214	47.7%
matlab	2390	1020	1.02	27496	8303	69.8%
Total			1.16			60.5%

Table 3. Various Slicing Approaches

	Full Slicing		Simple Std. PD		Naïve PD	
Benchmark	Time (ms)	Reduction	Time (ms)	Reduction	Time (ms)	Reduction
inter_medium	20	67.1%	50	67.1%	20	41.8%
lambdaint	390	35.5%	880	9.0%	30	9.0%
ctl_trace	1940	47.7%	140	47.7%	40	1.3%
matlab	2390	69.8%	1170	69.8%	200	19.3%
Total	4740	60.5%	2240	56.4%	290	17.0%

benchmark is the CTL model checker from [20], extended to compute witness traces. It is sliced for a particular system and temporal logic formula to model check. The `lambdaint` benchmark is an interpreter for a simple functional language taken from [17]. It is sliced for a particular functional program (computing the Fibonacci numbers). Finally, `matlab` is an interpreter for a subset of the Matlab language (the code can be found in [22]). The overall results are very good: the code size is reduced by 60.5% and runtime decreased by 16%.

Comparing the Influence of Local and Global Control. In Table 3, we compare the influence of the partial deduction control. Here, "Full slicing" is the standard CPD that we have used so far; "Simple Std. PD" is a standard (non-conjunctive) partial deduction with relatively simple control; and "Naïve PD" is very simple standard partial deduction in the style of [31], i.e., with a one-step unfolding and very simple generalization (although it is still more precise than [31] as it can produce some polyvariance), where we have turned the redundant argument filtering off.

The experiments we conducted (see [22] for the table of results) show the clear difference between our slicing approach and one using a naïve PD on the DPPD benchmarks used earlier: our approach manages a code size reduction of 26% whereas the naïve PD approach manages just 9.4%. The table also shows that the overall impact of the filtering is quite small. This is somewhat surprising, and may be due to the nature of the benchmarks. However, it may also mean that in the future we have to look at more powerful filtering approaches.

7 Discussion, Related and Future Work

In this work, we have introduced the first, semantics-preserving, forward slicing technique for logic programs. Traditional approaches to program slicing rely on the construction of some data structure to store the data and control dependences in a program. The key contribution of this paper has been to show that CPD can actually play such a role. The main advantages of this approach are the following: there is no need to distinguish between static and dynamic slicing and, furthermore, a slicing tool can be fully implemented with a modest implementation effort, since only the final code generation phase should be changed (i.e., the core algorithm of the partial deduction system remains untouched). A slicing tool has been fully implemented in ECCE, where a post-processing transformation to remove redundant arguments has been added. Our experiments demonstrate the usefulness of our approach, both as a classical slicing method as well as a technique for code size reduction.

As mentioned before, we are not aware of any other approach to forward slicing of logic programs. Previous approaches have only considered *backward* slicing. For instance, Schoening and Ducassé [27] defined the first backward slicing algorithm for Prolog which produces executable programs. Vasconcelos [30] introduced a flexible framework to compute both static and dynamic backward slices. Similar techniques have also been defined for constraint logic programs [28] and concurrent logic programs [33]. Within imperative programming, Field, Ramalingam, and Tip [6] introduced a *constrained* slicing scheme in which source programs are translated to an intermediate graph representation. Similarly to our approach, constrained slicing generalizes the traditional notions of static and dynamic slicing since arbitrary constraints on the input data can be made.

The closest approaches are those of [31] and [21]. Vidal [31] introduced a forward slicing method for lazy functional logic programs that exploits the similarities between slicing and partial evaluation. However, only a restrictive form of partial evaluation—i.e., monovariant and monogenetic partial evaluation—is allowed, which also restricts the precision of the computed slices. Our new approach differs from that of [31] in several aspects: we consider logic programs; we use a polyvariant and polygenetic partial evaluation scheme and, therefore, the computed slices are significantly more precise; and, moreover, since the basic partial deduction algorithm is kept unmodified, it can easily be implemented on top of an existing partial deduction system. On the other hand, Leuschel and Sørensen [21] introduced the concept of *correct erasure* in order to detect and remove redundant arguments from logic programs. They present a constructive algorithm for computing correct erasures which can be used to perform a simple form of slicing. In our approach, we use this algorithm as a post-processing phase to slice out unnecessary arguments of predicates in the computed slices. The combination of these two approaches, [31] and [21], together with a special-purpose slicing code generator, form the basis of a powerful forward slicing technique.

Since our work constitutes a first step towards the development of a forward slicing technique for logic programs, there are many interesting topics for future work. For instance, an interesting topic for further research involves the compu-

tation of *backward* slices (a harder topic). In this case, the information gathered by characteristic trees is not enough and some extension is needed.

One should also investigate to what extent abstract interpretation can be used to complement our slicing technique. On its own, abstract interpretation will probably lack the precise propagation of concrete values, hence making it less suitable for dynamic slicing. However, for static slicing it may be able to remove certain clauses that a partial deduction approach cannot remove (see, e.g., [4, 8] where useless clauses are removed to complement partial deduction) and one should investigate this possibility further. One could also investigate better global control, adapted for slicing (to avoid wasted specialisation effort in case added polyvariance does not increase the precision of the slice). Finally, we can use our slicing technique as a starting point for resource aware specialization, i.e., finding a good tradeoff between code size and execution speed.

Acknowledgements. We would like to thank the anonymous referees of ESOP'05 for their valuable feedback. We also would like to thank Dan Elphick, John Gallagher, Germán Puebla, and all the other partners of the ASAP project for their considerable help and input.

References

1. Advanced Specialization and Analysis for Pervasive Computing. EU IST FET Programme Project Number IST-2001-38059. `http://www.asap.ecs.soton.ac.uk/`
2. E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 2(1):3–26, 2002.
3. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 41(2 & 3):231–277, 1999.
4. D. A. de Waal and J. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 207–221. Springer-Verlag, 1994.
5. J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.
6. J. Field, G. Ramalingam, and F. Tip. Parametric Program Slicing In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 379-392, ACM Press, 1995.
7. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
8. J. Gallagher and D. A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proc. of LOPSTR'92*, pages 151–167, Manchester, UK, 1992.
9. J. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialisation. *New Generation Computing*, 9(3-4):305–333, 1991.
10. T. Gyimóthy and J. Paakki. Static Slicing of Logic Programs. In *Proc. of the 2nd Int'l Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, pages 87–103. IRISA-CNRS, 1995.

11. M. Harman and S. Danicic. Amorphous Program Slicing. In *Proc. of the 5th Int'l Workshop on Program Comprehension*. IEEE Computer Society Press, 1997.
12. M. Harman and R. Hierons. An Overview of Program Slicing. *Software Focus*, 2(3):85–92, 2001.
13. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
14. J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'96*, LNCS 1207, pages 59–82. Springer-Verlag, 1996.
15. M. Leuschel. The DPPD Library of Benchmarks. Accessible from URL: `http://www.ecs.soton.ac.uk/~mal/systems/dppd.html`
16. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
17. M. Leuschel, S. Craig, M. Bruynooghe, and W. Vanhoof. Specializing interpreters using offline partial deduction. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, pages 341–376. Springer LNCS 3049, 2004.
18. M. Leuschel and D. De Schreye. Constrained Partial Deduction and the Preservation of Characteristic Trees. *New Generation Computing*, 16(3):283–342, 1998.
19. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.
20. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, 2000.
21. M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'96*, LNCS 1207, pages 83–103. Springer-Verlag, 1996.
22. M. Leuschel and G. Vidal. Forward Slicing by Conjunctive Partial Deduction and Argument Filtering. Technical Report, DSSE, University of Southamtpon, December 2004.
23. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.
24. A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *The Journal of Logic Programming*, 19,20:261–320, 1994.
25. T. Reps and T. Turnidge. Program Specialization via Program Slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle*, pages 409–429. Springer LNCS 1110, 1996.
26. R. Glück and M.H. Sørensen A Roadmap to Metacomputation by Supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle*, pages 137–160. Springer LNCS 1110, 1996.
27. S. Schoenig and M. Ducassé. A Backward Slicing Algorithm for Prolog. In *Proc. of the Int'l Static Analysis Symposium (SAS'96)*, pages 317–331. Springer LNCS 1145, 1996.
28. G. Szilagyi, T. Gyimothy, and J. Maluszynski. Static and Dynamic Slicing of Constraint Logic Programs. *J. Automated Software Engineering*, 9(1):41–65, 2002.
29. F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

30. W. Vasconcelos. A Flexible Framework for Dynamic and Static Slicing of Logic Programs. In *Proc. of the First Int'l Workshop on Practical Aspects of Declarative Languages (PADL'99)*, pages 259–274. Springer LNCS 1551, 1999.
31. G. Vidal. Forward slicing of multi-paradigm declarative programs based on partial evaluation. In M. Leuschel, editor, *Proc. of Logic-based Program Synthesis and Transformation (LOPSTR'02)*, LNCS 2664, pages 219–237. Springer-Verlag, 2003.
32. M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.
33. J. Zhao, J. Cheng, and K. Ushijima. A Program Dependence Model for Concurrent Logic Programs and Its Applications. In *Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM'01)*, pages 672–681. IEEE Press, 2001.

A New Foundation for Control-Dependence and Slicing for Modern Program Structures*

Venkatesh Prasad Ranganath[1], Torben Amtoft[1], Anindya Banerjee[1], Matthew B. Dwyer[2], and John Hatcliff[1]

[1] Department of Computing and Information Sciences, Kansas State University**
{rvprasad, tamtoft, ab, hatcliff}@cis.ksu.edu
[2] Department of Computer Science and Engineering, University of Nebraska, Lincoln***
dwyer@cse.unl.edu

Abstract. The notion of control dependence underlies many program analysis and transformation techniques. Despite wide applications, existing definitions and approaches for calculating control dependence are difficult to apply seamlessly to modern program structures. Such program structures make substantial use of exception processing and increasingly support reactive systems designed to run indefinitely.

This paper revisits foundational issues surrounding control dependence and slicing. It develops definitions and algorithms for computing control dependence that can be directly applied to modern program structures. A variety of properties show that the new definitions conservatively extend classic definitions. In the context of slicing reactive systems, the paper proposes a notion of slicing correctness based on weak bisimulation and proves that the definition of control dependence generates slices that conform to this notion of correctness. The new definitions and algorithms for control dependence form the basis of a publicly available program slicer that has been implemented for full Java.

1 Introduction

The notion of control-dependence underlies many program analysis and transformation techniques used in numerous applications including program slicing applied for program understanding [2], debugging [3], partial evaluation [4], compiler optimizations [5] such as global scheduling, loop fusion, code motion etc. Intuitively, a program statement n_1 is control-dependent on a statement n_2, if n_2 (typically, a conditional statement) controls whether or not n_1 will be executed or bypassed during an execution of the program.

While existing definitions and approaches for calculating control dependence and slicing are widely applied and have been used in the current form for well over 20

* This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF (CCR-0306607,CCR-0296182,CCR-0209205,ITR-0326577,CCR-0444167), by Lockheed Martin, and by Intel Corporation.

** Manhattan KS, 66506, USA.

*** Lincoln NE 68588-0115, USA.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 77–93, 2005.

years, there are several aspects of these definitions that prevent them from being applied smoothly to modern program structures which rely significantly on exception processing and increasingly support reactive systems which are designed to run indefinitely.

(I.) Classic definitions of control dependence are stated in terms of program control-flow graphs (CFGs) in which the CFG has a unique end node – they do not apply directly to program CFGs with (a) multiple end nodes or with (b) no end node. Restriction (a) means that existing definitions cannot be applied directly to programs/methods with multiple exit points – a restriction that would be violated by any method that raises exceptions or includes multiple returns. Restriction (b) means that existing definitions cannot be applied directly to reactive programs or system models with control loops that are designed to run indefinitely.

Restriction (a) is usually addressed by performing a pre-processing step that transforms a CFG with multiple end nodes into a CFG with a single end node by adding a new designated end node to the CFG and inserting arcs from all original exit states to the new end node [6, 2]. Restriction (b) can also be addressed in a similar fashion by, e.g., selecting a single node within the CFG to represent the end node. This case is more problematic than the pre-processing for Restriction (a) because the criteria for selecting end nodes that lead to the desired control dependence relation between program nodes is often unclear. This is particularly true in threads such as event-handlers which have no explicit shut-down methods, but are "shut down" by killing the thread (thus, there is nothing in the thread's control flow to indicate an exit point).

(II.) A deeper problem is that existing definitions of slicing correctness either apply to programs with terminating execution traces, or they often fail to state whether or not the slicing transformation preserves the termination behavior of the program being sliced. Thus these definitions cannot be applied to reactive programs that are designed to execute indefinitely. Such programs are used in numerous modern applications such as event-processing modules in GUI systems, web services, distributed real time systems with autonomous components, e.g. data sensors, etc.

Despite the difficulties, it appears that researchers and practitioners do continue to apply slicing transformations to programs that fail to satisfy the restrictions above. However, in reality the pre-processing transformations related to issue **(I)** introduce extra overhead into the entire transformation pipeline, clutter up program transformation and visualization facilities, necessitate the use/maintenance of mappings from the transformed CFGs back to the original CFGs, and introduce extraneous structure with ad-hoc justifications that all down-stream tools/transformations must interpret and build on in a consistent manner. Moreover, regarding issue **(II)**, it will be infeasible to continue to ignore issues of termination as slicing is increasingly applied in high-assurance applications such as reducing models for verification [7] and for reasoning about security issues where it is crucial that liveness/non-termination properties be preserved.

Working on a larger project on slicing concurrent Java programs, we have found it necessary to revisit basic issues surrounding control dependence and have sought to develop definitions that can be directly applied to modern program structures such as those found in reactive systems. In this paper, we propose and justify the usefulness and correctness of simple definitions of control dependence that overcome the problematic

aspects of the classic definitions described above. The specific contributions of this paper are as follows.

- We propose new definitions of control dependence that are simple to state and easy to calculate and that work directly on control-flow graphs that may have no end nodes or non-unique end nodes, thus avoiding troublesome pre-processing CFG transformations (Section 4).
- We prove that these definitions applied to reducible CFGs yield slices that are correct according to generalized notions of slicing correctness based on a form of weak bisimulation that is appropriate for programs with infinite execution traces (Section 5.1).
- We clarify the relationship between our new definitions and classic definitions by showing that our new definitions represent a form of "conservative extension" of classic definitions: when our new definitions are applied to CFGs that conform to the restriction of a single end node, our definitions correspond to classic definitions – they do not introduce any additional dependences nor do they omit any dependences (Section 4.1).
- We discuss the intuitions behind algorithms for computing control dependence (according to the new definitions) to justify that control dependence is computable in polynomial time (Section 6).

Expanded discussions, definitions and full proofs appear in the companion technical report [8]. The proposed notions of control dependence described in this paper have been implemented in Indus [9] – our publicly available open-source Eclipse-based Java slicer that works on full Java 1.4 and has been applied to code bases of up to 10,000 lines of Java application code ($<$ 80K bytecodes) excluding library code. Besides its application as a stand-alone program visualization, debugging, and code transformation tool, our slicer is being used in the next generation of Bandera, a tool set for model-checking concurrent Java systems.[1]

2 Basic Definitions

2.1 Control Flow Graphs

In the sequel, we follow tradition and represent a program as a control-flow graph, whose definition we adapt from Ball and Horwitz [10].

Definition 1 (Control Flow Graphs).
A control-flow graph $G = (N, E, n_0)$ *is a labeled directed graph in which*

- *N is a set of nodes that represent commands in program,*
- *the set of N is partitioned into two subsets* N^S, N^P, *where* N^S *are* statement nodes *with each* $n_s \in N^S$ *having at most one successor, where* N^P *are* predicate nodes *with each* $n_p \in N^P$ *having two successors, and* $N^E \subseteq N^S$ *contains all nodes of* N^S *that have no successors, i.e.,* N^E *contains all end nodes of G,*

- *E is a set of labeled edges that represent the control flow between graph nodes where each* $n_p \in N^P$ *has two outgoing edges labeled* T *and* F *respectively, and each* $n_s \in (N^S - N^E)$ *has an outgoing edge labeled* A *(representing* Always taken*),*
- *the start node* n_0 *has no incoming edges and all nodes in N are reachable from* n_0.

We will display the labels on CFG edges only when necessary for the current exposition.

As stated earlier, existing presentations of slicing require that each CFG *G* satisfies the *unique end node property*: there is exactly one element in $N^E = \{n_e\}$ and n_e is reachable from all other nodes of *G*. The above definition *does not* require this property of CFGs, but we will sometimes consider CFGs with the unique end node property in our comparisons to previous work.

To relate a CFG with the program that it represents, we use the function *code* to map a CFG node *n* to the code for the program statement that corresponds to that node. Specifically, for $n_s \in N^S$, $code(n_s)$ yields the code for an assignment statement, and for $n_p \in N^P$, $code(n_p)$ the code for the test of a conditional statement (the labels on the edges for n_p allow one to determine the nodes for the true and false branches of the conditional). The function *def* maps each node to the set of variables defined (*i.e.*, assigned to) at that node (always a singleton or empty set), and *ref* maps each node to the set of variables referenced at that node.

A CFG *path* π from n_i to n_k is a sequence of nodes $n_i, n_{i+1}, \ldots, n_k$ such for every consecutive pair of nodes (n_j, n_{j+1}) in the path there is an edge from n_j to n_{j+1}. A path between nodes n_i and n_k can also be denoted as $[n_i..n_k]$. When the meaning is clear from the context, we will use π to denote the set of nodes contained in π and we write $n \in \pi$ when n occurs in the sequence π. Path π is *non-trivial* if it contains at least two nodes. A path is *maximal* if it is infinite or if it terminates in an end node.

The following definitions describe relationships between graph nodes and the distinguished start and end nodes [11]. Node n *dominates* node m in *G* (written *dom*(n, m)) if every path from the start node s to m passes through n (note that this makes the dominates relation reflexive). Node n *post-dominates* node m in *G* (written *post-dom*(n, m)) if every path from node m to the end node e passes through n. Node n *strictly post-dominates* node m in *G* if *post-dom*(n, m) and $n \neq m$. Node n is the *immediate post-dominator* of node m if $n \neq m$ and n is the first post-dominator on every path from m to the end node e. Note that domination relations are well-defined but post-domination relationships are not well-defined for graphs that do not have the unique end node property. Node n *strongly post-dominates* node m in *G* if n post-dominates m and there is an integer $k \geq 1$ such that every path from node m of length $\geq k$ passes through n [2]. The difference between strong post-domination and the simple definition of post-domination above is that even though node n occurs on every path from m to e (and thus n post-dominates m), it may be the case that there is a loop in the CFG between m and n that admits an infinite path beginning at m that never encounters n. Strong post-domination rules out the possibility of such loops between m and n – thus, it is sensitive to the possibility of non-termination along paths from m to n.

A CFG *G* of the form (N, E, n_0) is *reducible* if *E* can be partitioned into disjoint sets E_f (the *forward* edge set) and E_b (the *back* edge set) such that (N, E_f) forms a DAG in which each node can be reached from the entry node n_0 and for all edges $e \in E_b$, the target of e dominates the source of e. All "well-structured" programs, including Java

programs, give rise to reducible control-flow graphs. Our definitions and most of our correctness results apply to irreducible CFGs as well, but our correctness result of slicing based on bisimulation holds for reducible graphs since bisimulation requires ordering properties that can only be guaranteed on reducible graphs – (see example in Section 4 preceding Theorem 2.)

2.2 Program Execution

The execution semantics of program CFGs is phrased in terms of transitions on program states (n, σ) where n is a CFG node and σ is a store mapping the corresponding program's variables to values. A series of transitions gives an *execution trace* through p's statement-level control flow graph. For state (n_i, σ_i), the code at n_i is executed on the transition from (n_i, σ_i) to successor state (n_{i+1}, σ_{i+1}). Execution begins at the start node n_0, and the execution of each node possibly updates the store and transfers control to an appropriate successor node. Execution of a node $n_e \in N^E$ produces a final state (halt, σ) where the control point is indicated by a special label halt – this indicates a normal termination of program execution. The presentation of slicing in Section 5 involves arbitrary finite and infinite non-empty sequences of states written $\Pi = s_1, s_2, \ldots$. For a set of variables V, we write $\sigma_1 =_V \sigma_2$ when for all $x \in V$, $\sigma_1(x) = \sigma_2(x)$.

2.3 Notions of Dependence and Slicing

A *program slice* consists of the parts of a program p that (potentially) affect the variable values referenced at some program points of interest; such program points are traditionally called the *slicing criterion* [12]. A slicing criterion C for a program p is a non-empty set of nodes $\{n_1, \ldots, n_k\}$ where each n_i is a node in p's CFG.

The definitions below are the classic ones of the two basic notions of dependence that appear in slicing of sequential programs: *data dependence* and *control dependence* [12].

Data dependence captures the notion that a variable reference is dependent upon any variable definition that "reaches" the reference.

Definition 2 (data dependence). *Node n is* data-dependent *on m (written $m \xrightarrow{dd} n$ – the arrow pointing in the direction of data flow) if there is a variable v such that: (1) there exists a non-trivial path π in p's CFG from m to n such that for every node $m' \in \pi - \{m, n\}$, $v \notin def(m')$, and (2) $v \in def(m) \cap ref(n)$.*

Control dependence information identifies the conditionals that may affect execution of a node in the slice. Intuitively, node n is control-dependent on a predicate node m if m directly determines whether n is executed or "bypassed".

Definition 3 (control dependence). *Node n is* control-dependent *on m in program p (written $m \xrightarrow{cd} n$) if (1) there exists a non-trivial path π from m to n in p's CFG such that every node $m' \in \pi - \{m, n\}$ is post-dominated by n, and (2) m is not strictly post-dominated by n.*

For a node n to be control-dependent on predicate m, there must be two paths that connect m with the unique end node e such that one contains n and the other does not. There are several slightly different notions of control-dependence appearing in the literature, and we will consider several of these variants and relations between them in

the rest of the paper. At present, we simply note that the above definition is standard and widely used (e.g., see [11]).

We write $m \xrightarrow{d} n$ when either $m \xrightarrow{dd} n$ or $m \xrightarrow{cd} n$. The algorithm for constructing a program slice proceeds by finding the set of CFG nodes S_C (called the *slice set* or *backward static slice*) from which the nodes in C are reachable via $\xrightarrow{d}$. The term "backward" signifies that the algorithm starts at the criterion nodes and looks backward through the program's control-flow graph to find other program statements that influence the execution at the criterion nodes. Our definitions of control dependence can be applied in computing forward slices as well.

Definition 4 (slice set). *Let C be a slicing criterion for program p. Then the slice set S_C of p with respect to C is defined as follows:*

$$S_C = \{m \mid \exists n \,.\, n \in C \text{ and } m \xrightarrow{d}{}^{*} n\}.$$

We will consider slicing correctness requirements in greater detail in Section 5.1. For now we note that commonly in the slicing literature the desired correspondence between the source program and the slice is not formalized; the emphasis is often on applications rather than foundations, and this also leads to subtle differences between presentations. When a notion of "correct slice" is given, it is often stated using the notion of *projection* [13]. Informally, given an arbitrary trace Π of p and an analogous trace Π_s of p_s, p_s is a correct slice of p if projecting out the nodes in criterion C (and the variables referenced at those nodes) for both Π and Π_s yields identical state sequences.

3 Assessment of Existing Definitions

3.1 Variations in Existing Control Dependence Definitions

Although Definition 3 of control dependence is widely used, there are a number of (often subtle) variations appearing in the literature. Here are some:

Admissibility of indirect control dependences. For example, using the definition of control dependence in Definition 3, for Fig. 1 (a), we can conclude that $a \xrightarrow{cd} f$ and $f \xrightarrow{cd} g$ however $a \xrightarrow{cd} g$ does not hold because g does not post-dominate f. The fact that a and g are indirectly related (a does play a role in determining if g is executed or bypassed) is not captured in the definition of control dependence itself but in the transitive closure used in the slice set construction (Definition 4). However, some definitions of control dependence [2] incorporate this notion of transitivity directly into the definition itself as we will illustrate later.

Sensitivity to non-termination. Consider Fig. 1 (a) again, where node c represents a post-test that controls a potentially infinite loop. According to Definition 3, $a \xrightarrow{cd} d$ holds but $c \xrightarrow{cd} d$ does not hold (because d post-dominates c) even though c may determine whether d executes or never gets to execute due to an infinite loop that postpones d forever. Thus, Definition 3 is *non-termination insensitive*.

We now further illustrate the variations by recalling definitions of strong and weak control dependence given by Podgurski and Clarke [2] and used in a number of works, e.g., the study of control dependence by Bilardi and Pingali [14].

Definition 5 (Podgurski-Clarke Control Dependence).

- *n_2 is strongly control dependent on n_1 ($n_1 \overset{PC-scd}{\rightarrow} n_2$) if there is a path from n_1 to n_2 that does not contain the immediate post dominator of n_1.*
- *n_2 is weakly control dependent on n_1 ($n_1 \overset{PC-wcd}{\rightarrow} n_2$) if n_2 strongly post dominates n_1', a successor of n_1, but does not strongly post dominate n_1'', another successor of n_1.*

Whereas Definition 3 captures direct control dependence only, strong control dependence as defined above captures indirect control dependence. For example, in Fig. 1, in contrast to Definition 3, we have $a \overset{PC-scd}{\rightarrow} g$ because there is a path *afg* which does not contain e, the immediate post-dominator of *a*. However, one can show that when used in the context of Definition 4 (which computes the transitive closure of dependences), the two definitions give rise to the same slices.

Weak control dependence subsumes the notion of strong control dependence ($n_1 \overset{PC-scd}{\rightarrow} n_2$ implies $n_1 \overset{PC-wcd}{\rightarrow^*} n_2$) and it captures weaker dependences between nodes induced by non-termination: it is non-termination sensitive. For Fig. 1 (a), $c \overset{PC-wcd}{\rightarrow} d$ because *d* does not strongly post-dominate *b*: the presence of the loop controlled by *c* guarantees that there exists no k such that every path from node *b* of length $\geq k$ passes through *d*.

The impact of the variations on slicing. Note that slicing based on Definition 3 or the strong control dependence above can transform a non-terminating program into a terminating one (i.e., non-termination is not preserved in the slice). In Fig. 1 (a), assume that the loop controlled by *c* is an infinite loop. Using the slice criterion $C = \{d\}$ would include *a* but not *b* and *c* (we assume no data dependence between *d* and *b* or *c*) if the slicing is based on strong control dependence. Thus, in the sliced program, one would be able to observe an execution of *d*, but such an observation is not possible in the original program because execution diverges before *d* is reached. In contrast, the difference between direct and indirect statements of control dependence seems to be largely a technical stylistic decision in how the definitions are stated.

Very few works consider the non-termination sensitive notion of weak control dependence above. We conjecture that there are at least two reasons for this. First, weak control dependence is actually a larger relation (relating more nodes) and will thus include more nodes in the slice. Second, many applications of slicing focus on debugging and program visualization and understanding, and in these applications having slices that preserve non-termination is less important than having smaller slices. However, slicing is increasingly used in security applications and as a model-reduction technique for software model checking. In these applications, it is important to consider variants of control dependence that preserve non-termination properties, since failure to do so could allow inferences to be made that compromise security policies, for instance invalidate checks of liveness properties [7].

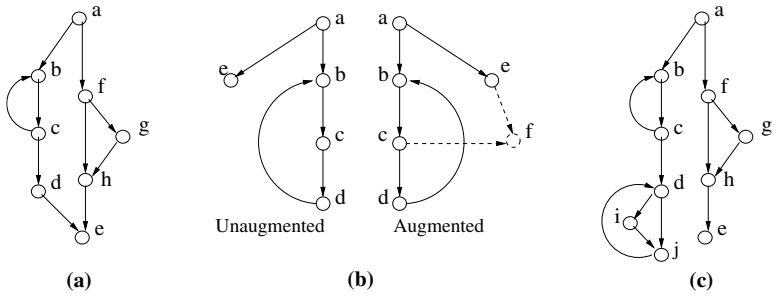


Fig. 1. (a) is a simple CFG. (b) illustrates how a CFG that does not have a unique exit node reachable from all nodes can be augmented to have unique exit node reachable from all nodes. (c) is a CFG with multiple control sinks of different sorts

3.2 Unique End Node Restriction on CFGs

All definitions of control dependences that we are aware of require that CFGs satisfy the unique end node requirement – but many software systems fail to satisfy this property. Existing works simply require that CFGs have this property, or they suggest that CFGs can be augmented to achieve this property, e.g., using the following steps: (1) insert a new node *e* into the CFG, (2) add an edge from each exit node (other than *e*) to *e*, (3) pick an arbitrary node *n* in each non-terminating loop and add an edge from *n* to *e*.

In our experience, such augmentations complicate the system being analyzed in several ways. If the augmentation is non-destructive, a new CFG is generated which costs time and memory. If the augmentation is destructive, this may clash with the requirements of other clients of the CFG, thus necessitating the reversal of the augmentation before subsequent analyses can proceed. If the augmentation is not reversed, the graph algorithms and analyses algorithms should be made intelligent to operate on the actual CFG embedded in the augmented CFG.

Many systems have threads where the main control loop has no exit – the loop is "exited" by simply killing the thread. For example, in Xt library, most applications create widgets, register callbacks, and call `XtAppMainLoop()` to enter an infinite loop that manages the dispatching of events to the widgets in the application. In PalmOS, applications are designed such that they start upon receiving a start code, execute a loop, and terminate upon receiving a stop code. However, the application may choose to ignore the stop code once it starts, and hence, not terminate except when it is explicitly killed. In such cases, a node in the loop must be picked as the loop exit node for the purpose of augmenting the CFG. But this can disrupt the control dependence calculations. In Fig. 1 (b), we would intuitively expect e,b,c, and d to be control dependent on a in the unaugmented CFG. However, $a \overset{PC-wcd}{\rightarrow} \{e, b, c\}$ and $c \overset{PC-wcd}{\rightarrow} \{b, c, d, f\}$ in the augmented CFG. It is trivial to prune dependences involving *f*. But now there are new dependences $c \overset{PC-wcd}{\rightarrow} \{b, c, d\}$ which did not exist in the unaugmented CFG. Although a suggestion to delete any dependence on *c* may work for the given CFG, it fails if there exists a node *g* that is a successor of *c* and a predecessor of *d*. Also, $a \overset{PC-wcd}{\rightarrow} d$ exists in the unaugmented CFG but not in the augmented CFG, and it is not obvious how to recover this information.

We address these issues head-on by considering alternate definitions of control-dependence that do not impose the unique end-node description.

4 New Dependence Definitions

In previous definitions, the control dependence of n_j on n_i is specified by considering paths from n_i and n_j to a unique CFG end node – essentially n_i and the end node delimit the path segments that are considered. Since we aim for definitions that apply when CFGs do not have an end node or have more than one end node, we aim to instead specify that n_j is control dependent on n_i by focusing on paths between n_i and n_j. Specifically, we focus on path segments that are delimited by n_i *at both ends* – intuitively corresponding to the situation in a reactive program where instead of reaching an end node, a program's behavior begins to repeat itself by returning again to n_i. At a high level, the intuition remains the same as in, e.g., Definition 3 – executing one branch of n_i always leads to n_j, whereas executing another branch of n_i can cause n_j to be bypassed. The additional constraints that are added (e.g., n_j always occurs before any occurrence of n_i) limits the region in which n_j is seen or bypassed to segments leading up to the next occurrence of n_i – ensuring that n_i is indeed *controlling* n_j. The definition below considers maximal paths (which includes infinite paths) and thus is sensitive to non-termination.

Definition 6 ($n_i \overset{ntscd}{\rightarrow} n_j$). *In a CFG, n_j is **(directly) non-termination sensitive control dependent** on node n_i, if n_i has at least two successors, n_k and n_l, and (1) for all maximal paths from n_k, n_j always occurs and, either $n_j = n_i$, or n_j occurs before any occurrence of n_i; and, (2) there exists a maximal path from n_l on which either n_j does not occur, or n_j is strictly preceded by n_i.*

We supplement a traditional presentation of dependence definitions with definitions given as formulae in computation tree logic (CTL) [15]. CTL is a logic for describing the structure of sets of paths in a graph, making it a natural language for expressing control dependences. Informally, CTL includes two path quantifiers, E and A, which define that a path from a given node with a given structure exists or that all paths from that node have the given structure. The structure of a path is defined using one of five modal operators (we refer to a node satisfying ϕ as a ϕ-node): $\mathsf{X}\phi$ states that the successor node is a ϕ-node, $\mathsf{F}\phi$ states the existence of a ϕ-node, $\mathsf{G}\phi$ states that a path consists entirely of ϕ-nodes, $\phi\mathsf{U}\psi$ states the existence of a ψ-node and that the path leading up to that node consists of ϕ-nodes, finally, the $\phi\mathsf{W}\psi$ operator is a variation on U that relaxes the requirement that a ψ-node exist. In a CTL formula path quantifiers and modal operators occur in pairs, e.g., $\mathsf{AF}\phi$ says on all paths from a node a ϕ node occurs.

The following CTL formula captures the definition of control dependence above.

$$n_i \overset{ntscd}{\rightarrow} n_j = (G, n_i) \models \mathsf{EX}(\mathsf{A}[\neg n_i \mathsf{U} n_j]) \wedge \mathsf{EX}(\mathsf{E}[\neg n_j \mathsf{W}(\neg n_j \wedge n_i)])$$

Here, $(G, n_i) \models$ expresses the fact that the CTL formula is checked against the graph G at node n_i. The two conjuncts are essentially a direct transliteration of the two conditions in Definition 6.

We have formulated Definition 6 to apply to *execution traces* instead of CFG paths. In this setting one needs to bound relevant segments by n_i as discussed above. However,

when working on CFG paths, the definition conditions can actually be simplified to read as follows: (1) *for all maximal paths from n_k, n_j always occurs*, and (2) *there exists a maximal path from n_l on which n_j does not occur*. A CTL formula for this simplified definition is

$$n_i \overset{ntscd}{\rightarrow} n_j = (G, n_i) \models \mathsf{EX}(\mathsf{AF}(n_j) \wedge \mathsf{EX}(\mathsf{EG}(\neg n_j))).$$

See [8] for the proof that the simplified definition and Definition 6 are equivalent on CFGs.

Illustrating non-termination sensitivity of Definition 6: Note that $c \overset{ntscd}{\rightarrow} d$ in Fig. 1 (a) since there exists a maximal path (an infinite loop between b and c) where d never occurs. In Fig. 1 (c), note that $d \overset{ntscd}{\rightarrow} i$ because there is an infinite path from j (cycle on (j,d)) on which i does not occur.

We now turn to constructing a non-termination insensitive version of control dependence. The definition above considered all paths leading out of a conditional. Now, we need to limit the reasoning to finite paths that reach a terminal region of the graph. To handle this in the context of CFGs that do not have the unique end-node property, we generalize the concept of *end node* to *control sink* – a set of nodes such that each node in the set is reachable from every other node in the set and there is no path leading out of the set. More precisely, a *control sink* κ is a set of CFG nodes that form a strongly connected component such that for each $n \in \kappa$ each successor of n is also in κ. It is trivial to see that each end node forms a control sink and each loop without any exit edges in the graph forms a control sink. For example, $\{e\}$ and $\{b, c, d\}$ are control sinks in Fig. 1 (b unaugmented), and $\{e\}$ and $\{d, i, j\}$ are control sinks in Fig. 1 (c). Let the set of *sink-bounded paths from n_k* (denoted *SinkPaths*(n_k)) contain all paths π from n_k to a node n_s such that n_s belongs to a control sink.

Definition 7 ($n_i \overset{nticd}{\rightarrow} n_j$). *In a CFG, n_j is **(directly) non-termination insensitively control dependent** on n_i if n_i has at least two successors, n_k and n_l, and (1) for all paths $\pi \in$ SinkPaths(n_k), $n_j \in \pi$; and, (2) there exists a path $\pi \in$ SinkPaths(n_l) such that $n_j \notin \pi$ and if π leads to a control sink κ, $n_j \notin \kappa$.*

In CTL:

$$n_i \overset{nticd}{\rightarrow} n_j = (G, n_i) \models \mathsf{EX}(\hat{\mathsf{A}}\mathsf{F}(n_j)) \wedge \mathsf{EX}(\hat{\mathsf{E}}[\neg n_j \mathsf{U}(\textit{c-sink?} \wedge n_j \notin \textit{c-sink})])$$

where: $\hat{\mathsf{A}}$ and $\hat{\mathsf{E}}$ represent quantification over sink-bounded paths only; *c-sink?* evaluates to *true* only if the current node belongs to a control sink; *c-sink* returns the sink set associated with the current node.

Illustrating non-termination insensitivity of Definition 7: Note that $c \overset{nticd}{\not\rightarrow} d$ in Fig. 1 (a) since all paths from c to the control sink, $\{e\}$, contain d. In Fig. 1 (b unaugmented) $a \overset{nticd}{\rightarrow} e$ because there exists a path from b to the control sink $\{b, c, d\}$ and neither the path nor the sink contain e; and, $a \overset{nticd}{\rightarrow} \{b, c, d\}$ because there is a path ending in control sink $\{e\}$ that does not contain b, c, or d. Interestingly, for Fig. 1 (c) our definition concludes that $d \overset{nticd}{\not\rightarrow} i$ because although there is a trivial path from d to the control sink $\{d, i, j\}$,

i belongs to that control sink. This is because the definition inherently captures a form of fairness – since the back edge from j guarantees that d will be executed an infinite number of times, the only way to avoid executing i would be to branch to d on every cycle. Consequently, even though there may be control structures inside of a control sink, they will not give rise to any control dependences. In applications where one desires to detect such dependences, one would apply the definition to control sinks in isolation with back edges removed.

4.1 Properties of the Dependence Relations

We begin by showing that the new definitions of control dependence conservatively extend classic definitions: when we consider our definitions in the original setting with CFGs with unique end nodes, the definitions coincide with the classic definitions. In addition, direct non-termination insensitive control dependence (Definition 7) implies the *transitive closure* of direct non-termination sensitive control dependence.

Theorem 1 (Coincidence Properties). *For all CFGs with the unique end node property, and for all nodes $n_i, n_j \in N$ we have: (1) $n_i \xrightarrow{cd} n_j$ implies $n_i \xrightarrow{nticd} n_j$; (2) $n_i \xrightarrow{nticd} n_j$ implies $n_i \xrightarrow{cd} n_j$; (3) $n_i \xrightarrow{PC-wcd} n_j$ iff $n_i \xrightarrow{ntscd} n_j$; (4) For all CFGs, for all nodes $n_i, n_j \in N : n_i \xrightarrow{nticd} n_j$ implies $n_i \xrightarrow{ntscd^*} n_j$.*

Part(4) of the above theorem is illustrated as follows: in Fig. 1 (a), $a \xrightarrow{nticd} d$ holds but $a \xrightarrow{ntscd} d$ does not. But $a \xrightarrow{ntscd^*} d$ holds as both $a \xrightarrow{ntscd} c$ and $c \xrightarrow{ntscd} d$ hold.

For the (bisimulation-based) correctness proof in Section 5.1, we shall need a few results about slice sets (members of which are termed "observable"). The main intuition is that the nodes in a slicing criteria C represent "observations" that one is making about a CFG G under consideration. Specifically, for an $n \in C$, one can observe that n has been executed and also observe the values of any variables referenced at n. A crucial property is that the first observable node on any path (n_1 in the lemmas below) will be encountered sooner or later on all other paths. Letting Ξ be the set of nodes, we have:

Lemma 1. *Assume Ξ is closed under $\xrightarrow{ntscd}$, and that $n_0 \notin \Xi$. Assume that there is a path π from n_0 to n_1, with $n_1 \in \Xi$ but for all $n \in \pi$ with $n \neq n_1$, $n \notin \Xi$. Then all maximal paths from n_0 will contain n_1.*

The notion of "closed" Ξ is this: if $n_i \in \Xi$ and $n_i \xrightarrow{ntscd} n_j$ then $n_j \in \Xi$.

Lemma 2. *Assume Ξ is closed under $\xrightarrow{nticd}$, and that $n_0 \notin \Xi$. Assume that there is a path π from n_0 to n_1, with $n_1 \in \Xi$ but for all $n \in \pi$ with $n \neq n_1$, $n \notin \Xi$. Then all sink-bounded paths from n_0 will contain n_1.*

As a consequence we have the following result, giving conditions to preclude the existence of infinite un-observable paths:

Lemma 3. *Assume that $n_0 \notin \Xi$, but that there is a path π starting at n_0 which contains a node in Ξ. (1) If Ξ is closed under $\xrightarrow{nticd}$, then all sink bounded paths starting at n_0 will reach Ξ. (2) If Ξ is also closed under $\xrightarrow{ntscd}$, then all maximal paths starting at n_0 will reach Ξ.*

We are now ready for the section's main result: from a given node there is a unique first observable. For this, we need the CFG to be reducible, as can be seen by the counterexample where from n_0 there are edges to n_1 and n_2 between which there is a cycle.

Theorem 2. *Assume that $n_0 \notin \Xi$, that $n_1, n_2 \in \Xi$, and that there are paths $\pi_1 = [n_0..n_1]$ and $\pi_2 = [n_0..n_2]$ such that on both paths, all nodes except the last do not belong to Ξ. If Ξ is closed under $\overset{ntscd}{\rightarrow}$ and if the CFG is reducible, then $n_1 = n_2$.*

5 Slicing

We now describe how to slice a (reducible) CFG G wrt. a slice set S_C, the smallest set containing C which is closed under data dependence $\overset{dd}{\rightarrow}$ and also closed under $\overset{ntscd}{\rightarrow}$.

The result of slicing is a program with the same CFG as the original one, but with the code map $code_1$ replaced by $code_2$. Here $code_2(n) = code_1(n)$ for $n \in S_C$; for $n \notin S_C$:

- if n is a statement node then $code_2(n)$ is the statement `skip`;
- if n is a predicate node then $code_2(n)$ is `cskip`, the semantics of which is that it non-deterministically chooses one of its successors.

The above definition is conceptually simple, so as to facilitate the correctness proofs. Of course, one would want to do some post-processing, like eliminating `skip` commands and eliminating `cskip` commands where the two successor nodes are equal; we shall not address this issue further but remark that most such transformations are trivially meaning preserving.

5.1 Correctness Properties

For a slicing criterion C, execution of nodes not in C correspond to *silent moves* or non-observable actions. The slicing transformation should preserve the behavior of the program with respect to C observations, but parts of the program that are irrelevant with respect to computing C observations can be "sliced away". The slice set S_C built according to Definition 4 represents the nodes that are relevant for maintaining the observations C. Thus, to prove the correctness of slicing we will establish the stronger result that G will have the same S_C observations wrt. the original code map $code_1$ as wrt. the sliced code map $code_2$, and this will imply that they have the same C observations.

The discussion above suggests that appropriate notions of correctness for slicing reactive programs can be derived from the notion of weak bisimulation found in concurrency theory, where a transition may include a number of τ-moves [16]. In our setting, we shall consider transitions that do one or more steps before arriving at a node in the slice set.

Definition 8. *For $i = 1, 2$, wrt. code map $code_i$: $s \overset{i}{\longmapsto} s'$ denotes that program state s rewrites in one step to s'. And, $s_0 \overset{i}{\Longrightarrow} s$ denotes that there exists $s_1 \ldots s_k$ ($k \geq 1$) with $s_k = s$ such that*
(1) for all $j \in \{1 \ldots k\}$ we have $s_{j-1} \overset{i}{\longmapsto} s_j$;
(2) $n_k \in S_C$ but for all $j \in \{1 \ldots k-1\}$, $n_j \notin S_C$, where $s_j = (n_j, \sigma_j)$ for each j.

Definition 9. *Binary relation $\mathcal{S}$ on program states is a bisimulation if whenever $(s_1, s_2) \in \mathcal{S}$ then: (a) if $s_1 \stackrel{1}{\Longrightarrow} s_1'$ then there exists s_2' such that $s_2 \stackrel{2}{\Longrightarrow} s_2'$ and $(s_1', s_2') \in \mathcal{S}$; and, (b) if $s_2 \stackrel{2}{\Longrightarrow} s_2'$ then there exists s_1' such that $s_1 \stackrel{1}{\Longrightarrow} s_1'$ and $(s_1', s_2') \in \mathcal{S}$.*

For each node *n* in *G*, we define *relv*(n), the set of relevant variables at *n*, by stipulating that $x \in$ *relv*(n) if there exists a node $n_k \in S_C$ and a path π from *n* to n_k such that $x \in$ *refs*(n_k), but $x \notin$ *defs*(n_j) for all nodes n_j occurring before n_k in π.

The above is well-defined in that it does not matter whether we use $code_1$ or $code_2$, as it is easy to see that the value of *relv*(n) is not influenced by the content of nodes not in S_C, since that set is closed under $\stackrel{dd}{\rightarrow}$. (Also, the closedness properties of S_C are not affected by using $code_2$ rather than $code_1$.) We have now arrived at the correctness theorem:

Theorem 3. *Let relation $\mathcal{S}_0$ be given by $(n_1, \sigma_1)\, \mathcal{S}_0\, (n_2, \sigma_2)$ iff $n_1 = n_2$ and $\sigma_1 =_{relv(n_1)} \sigma_2$. Then (given reducible G) if S_C is closed under $\stackrel{ntscd}{\rightarrow}$ then $\mathcal{S}_0$ is a bisimulation.*

6 Non-termination Sensitive Control Dependence Algorithm

Control dependences are calculated using a symbolic data-flow analysis. Each outgoing edge $n \rightarrow p$ of a predicate node n is represented by a token t_{np}. At each node m, a summary set S_{mn} is maintained for each predicate node n. Tokens are injected into the summary sets of the successors of each predicate node. The tokens are then propagated according to the following rules until no propagatian can occur.

- If q is a non-predicate node in $q \rightarrow r$ then the tokens in the summary sets at q are copied into the corresponding summary sets at r. This records that all maximal paths containing q also contain r.
- Only if all tokens corresponding to a predicate node n have arrived at node q then the tokens in the summary sets at n are copied into corresponding summary sets at q. This records that all maximal paths containing n also contain q.

Upon termination, $t_{np} \in S_{mn}$ indicates that all maximal paths from n starting with $n \rightarrow p$ contain m. Based on this observation, if $|S_{mn}| > 0 \wedge |S_{mn}| \neq T_n$ then, by Definition 6, it can be inferred that m *is directly control dependent on* n. On the other hand, if $|S_{mn}| > 0$ and $|S_{mn}| = T_n$ then, by Definition 6, it can be inferred that m *is not directly control dependent on* n.

The above algorithm has a worst-case asymptotic complexity of $O(|N|^3 \times K)$ where *K* is the sum of the outdegree of all predicate nodes in the CFG. Linear time algorithms to calculate control dependence based on augmented CFGs have been proposed in the literature [2]. The practical cost of this augmentation varies with the specific algorithm and the nature of control dependence being calculated. Our experience with an implementation of our algorithm in a program slicer for full Java [17] suggests that, despite its complexity bound, it elegantly scales to programs with tens-of-thousands of lines of code. We suspect that this is due in part to the elimination of the processing overhead involved in dealing with augmented CFGs.

```
NON-TERMINATION-SENSITIVE-CONTROL-DEPENDENCE(G)
 1  G(N,E,n_0,N^E) : a control flow graph.
 2  S[|N|,|N|] : a matrix of sets where S[n_1,n_2] represents S_{n_1 n_2}.
 3  T[|N|] : a sequence of integers where T[n_1] denotes T_{n_1}.
 4  CD[|N|] : a sequence of sets.
 5  workbag : a set of nodes.
 6
 7  # (1) Initialize
 8  workbag ← ∅
 9  for each n_1 in condNodes(G) and n_2 in succs(n_1,G)
10  do workbag ← workbag ∪ {n_2}
11     S_{n_2 n_1} ← {t_{n_1 n_2}}
12
13  # (2) Calculate all-path reachability
14  while workbag ≠ ∅
15  do flag ← false
16     n_3 ← remove(workbag)
17     for each n_1 in condNodes(G)\n_3
18     do if |S_{n_3 n_1}| = T_{n_1}
19           then for each n_4 in condNodes(G)\n_3
20                do if S_{n_1 n_4}\S_{n_3 n_4} ≠ ∅
21                   then S_{n_3 n_4} ← S_{n_3 n_4} ∪ S_{n_1 n_4}
22                        flag = true
23
24     if flag and |succs(n_3,G)| = 1
25       then n_5 ← the successor of n_3 in G
26            for n_4 in condNodes(G)
27            do if S_{n_5 n_4}\S_{n_3 n_4} ≠ ∅
28                 then S_{n_5 n_4} ← S_{n_5 n_4} ∪ S_{n_3 n_4}
29                      workbag ← workbag ∪ {n_5}
30       else if flag and |succs(n_3,G)| > 1
31              then for each n_4 in N
32                   do if |S_{n_4 n_3}| = T_{n_3}
33                      then workbag ← workbag ∪ {n_4}
34
35  # (3) Calculate non-termination sensitive control dependence
36  for each n_3 in N
37  do for each n_1 in condNodes(G)
38     do if |S_{n_4 n_3}| > 0 and |S_{n_3 n_1}| ≠ T_{n_1}
39           then CD[n_3] ← CD[n_3] ∪ {n_1}
40
41  return CD
```

Fig. 2. The algorithm to calculate non-termination sensitive control dependence

A complete description of the algorithm, its correctness, and its complexity analysis is given in [8].

7 Related Work

Fifteen years ago, control dependence was rigorously explored by Podgurski and Clarke [2]. Since then there has been a variety of work related to calculation and application of control dependence in the setting of CFGs that satisfy the unique end node property.

In the realm of calculating control dependence, Bilardi et.al [14] proposed new concepts related to control dependence along with algorithms based on these concepts to efficiently calculate weak control dependence. Johnson proposed an algorithm that could be used to calculate control dependence in time linear in the number of edges [18]. In comparison, in this paper we sketch a feasible algorithm in a more general setting.

In the context of slicing, Horwitz, Reps, and Binkley [19] presented what has now become the standard approach to inter-procedural slicing via dependence graphs. Recently, Allen and Horwitz [20] extended previous work on slicing to handle exception-based inter-procedural control flow. In this work, they handle CFG's with two end nodes (one for normal return and one for exceptional return) but it is unclear how this affects the control dependence captured by the dependence graph. In comparison, we have shown that program slicing is feasible with unaugmented CFGs.

For relevant work on slicing correctness, Horwitz et.al. use a semantics based multi-layered approach to reason about the correctness of slicing in the realm of data dependence [21]. Ball et.al use a program point specific history based approach to prove the correctness of slicing for arbitrary control flow [10]. We extend that work to consider arbitrary control flow without the unique end-node restriction. Their correctness property is a weaker property than bisimulation – it does not require ordering to be maintained between observable nodes if there is no dependence between these nodes – and it holds for irreducible CFGs. Even though our definitions apply to irreducible graphs, we need reducible graphs to achieve the stronger correctness property. We are currently investigating if we can establish their correctness property using our control dependence definitions on irreducible graphs.

Hatcliff et.al. present notions of dependence for concurrent CFGs, and propose a notion of bisimulation as the correctness property [6]. Millett and Teitelbaum [22] study static slicing of Promela (the model description language for the model-checker SPIN) and its application to model checking, simulation, and protocol understanding. They reuse existing notions of slicing, however, they neither discuss issues related to preservation of non-termination and liveness properties nor formalize a notion of correct slice for their applications. Krinke [23] considers static slicing of multi-threaded programs with shared variables, and focuses on issues associated with inter-thread data dependence but does not consider non-termination sensitive control dependence.

8 Conclusion

The notion of control dependence is used in myriad of applications, and researchers and tool builders increasingly seek to apply it to modern software systems and high-assurance applications – even though the control flow structure and semantic behavior of these systems do not mesh well with the requirements of existing control dependence definitions. In this paper, we have proposed conceptually simple definitions of control

dependence that (a) can be applied directly to the structure of modern software thus avoiding unsystematic preprocessing transformations that introduce overhead, conceptual complexity, and sometimes dubious semantic interpretations, and (b) provide a solid semantic foundation for applying control dependence to reactive systems where program executions may be non-terminating.

We have rigorously justified these definitions by providing detailed proofs of correctness (see the companion technical report [8]), by expressing them in temporal logic (which provides an unambiguous definition and allows them to be mechanically checked/debugged against examples using automated verification tools), by showing their relationship to existing definitions, and by implementing and experimenting with them in a publicly available slicer for full Java. In addition, we have provided algorithms for computing these new control dependence relations, and argued that any additional cost in computing these relations is negligible when one considers the cost and ill-effects of preprocessing steps required for previous definitions. Thus, we believe that there are many benefits for widely applying these definitions in static analysis tools.

In ongoing work, we continue to explore the foundations of statically and dynamically calculating dependences for concurrent Java programs for slicing, program verification, and security applications. In particular, we are exploring the relationship between dependences extracted from execution traces and dependences extracted from control-flow graphs in an effort to systematically justify a comprehensive set of dependence notions for the rich features found in concurrent Java programs. This effort will yield a more direct *semantic* connection between notions of dependence and execution traces instead of working indirectly through syntactic-oriented CFG definitions. With the translated, temporal logic-based dependence definitions, we are investigating how certain temporal properties of the unsliced version of the program are preserved in the sliced version.

References

1. Corbett, J., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby, Zheng, H.: Bandera: Extracting Finite-state Models from Java source code. In: 22^{nd} International Conference on Software Engineering (ICSE'00). (2000) 439–448.
2. Podgurski, A., Clarke, L.: A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Trans. Soft. Engg. **16** (1990) 965–979.
3. Francel, M.A., Rugaber, S.: The relationship of slicing and debugging to program understanding. In: Seventh IEEE International Workshop on Program Comprehension (IWPC'99). (1999) 106–113.
4. Anderson, L.O.: Program Analysis and Specialization for the C Programming Languages. PhD thesis, DIKU, University of Copenhagen (1999).
5. Ferrante, J., Ottenstein, K.J., Warren, J.O.: The program dependence graph and its use in optimization. ACM TOPLAS **9** (1987) 319–349.
6. Hatcliff, J., Corbett, J.C., Dwyer, M.B., Sokolowski, S., Zheng, H.: A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In: International Static Analysis Symposium (SAS'99) (1999), 1–18.
7. Hatcliff, J., Dwyer, M.B., Zheng, H.: Slicing software for model construction. Journal of Higher-order and Symbolic Computation **13** (2000) 315–353.

8. Ranganath, V.P., Amtoft, T., Banerjee, A., Dwyer, M.B., Hatcliff, J.: A new foundation for control-dependence and slicing for modern program structures. Technical Report 8, SAnToS Lab., Kansas State University (2004). Available at http://projects.cis.ksu.edu/docman/admin/index.php?editdoc=1&docid=95&group_id=12.
9. SAnToS Laboratory, Kansas State University: Indus, a toolkit to customize and adapt Java programs. Available at http://indus.projects.cis.ksu.edu.
10. Ball, T., Horwitz, S.: Slicing programs with arbitrary control-flow. In: First International Workshop on Automated and Algorithmic Debugging (AADEBUG). Volume 749 of Lecture Notes in Computer Science, Springer-Verlag (1993) 206–222.
11. Muchnick, S.S.: Advanced Compiler Design & Implementation. Morgan Kaufmann Publishers. Inc., San Francisco, California, USA (1997).
12. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages **3** (1995) 121–189.
13. Weiser, M.: Program slicing. IEEE Trans. Soft. Engg. **10** (1984) 352–357.
14. Bilardi, G., Pingali, K.: A framework for generalized control dependences. In: PLDI'96, 1996, 291–300.
15. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999).
16. Milner, R.: Communication and Concurrency. Prentice Hall (1989).
17. Jayaraman, G., Ranganath, V.P., Hatcliff, J.: Kaveri: Delivering Indus Java Program Slicer to Eclipse. In: Fundamental Approaches to Software Engineering (FASE'05), 2005. To appear.
18. Johnson, R., Pingali, K.: Dependence-based program analysis. In: PLDI'93, 1993, 78–89.
19. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM TOPLAS, 1990, 35–46.
20. Allen, M., Horwitz, S.: Slicing Java programs that throw and catch exceptions. In: PEPM'03, 2003, 44–54.
21. Horwitz, S., Pfeiffer, P., Reps, T.W.: Dependence analysis for pointer variables. In: PLDI'89, 1989, 28–40.
22. Millett, L., Teitelbaum, T.: Slicing Promela and its applications to model checking, simulation, and protocol understanding. In: Fourth International SPIN Workshop. (1998)
23. Krinke, J.: Static slicing of threaded programs. In: Workshop on Program Analysis for Software Tools and Engineering (PASTE'98). (1998) 35–42.

Summaries for While Programs with Recursion

Andreas Podelski, Ina Schaefer, and Silke Wagner

Max-Planck-Institut für Informatik, Saarbrücken, Germany

Abstract. Procedure summaries are an approximation of the effect of a procedure call. They have been used to prove partial correctness and safety properties. In this paper, we introduce a generalized notion of procedure summaries and present a framework to verify total correctness and liveness properties of a general class of while programs with recursion. We provide a fixpoint system for computing summaries, and a proof rule for total correctness of a program given a summary. With suitable abstraction methods and algorithms for efficient summary computation, the results presented here can be used for the automatic verification of termination and liveness properties for while programs with recursion.

1 Introduction

Procedure summaries are a fundamental notion in the analysis and verification of recursive programs [21, 19, 3]. They refer to the approximation of the "functional" effect of a procedure call. So far, they have shown useful for deriving and proving partial correctness, invariance and safety properties ("nothing bad may happen"). The results in this paper show that procedure summaries may be useful for deriving and proving termination and liveness properties ("something good will happen").

More specifically, we present a notion of summaries that applies to general programs with arbitrary nesting of while loops and recursion; the program variables range over possibly infinite data domains. A summary captures the effect of the unbounded unwinding of the body of procedure definitions, as well as of while loops. More generally, a summary may refer to any pair of programs points and captures the effect of computations that start and end at these program points.

We may use a pair of state assertions to express a summary, e.g. the pair $(x > 0, x < 0)$ to describe that the program variable x is first positive and then negative. We also may use assertions on state pairs, e.g. the assertion $x' = -x$ to describe that the program variable x gets multiplied by -1.

It is obvious that partial correctness and invariance and safety properties can be expressed in terms of summaries. This paper shows that also termination can be expressed in terms of summaries. We here concentrate on termination; the reduction of more general liveness properties to termination would follow the lines of [23, 14, 15].

The two classical proof rules for partial correctness and termination use invariants and variants (ranking functions) for the auxiliary assertion on the program. We present a proof rule for total correctness that uses summaries for the (one) auxiliary assertion on the program. Besides illustrating a new facet of total correctness of recursive programs, the contribution of the proof rule lies in its potential for automation via abstract interpretation [8, 9]. The considerable investment of research into the efficient computation of

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 94–107, 2005.

summaries has been a success; its payoff through industrialized tools checking invariance and safety properties of recursive programs [3] may well extend to termination and liveness properties. We believe that our paper may lead to several directions of follow-up work towards that goal.

2 Related Work

Among the vast amount of work on the analysis and verification of recursive programs, we will cover the part that seems most relevant for ours. In short, to advance a sum-up of the comparison, none of that work considers a notion of summary as general as ours (which refers to arbitrarily precise descriptions of the effect of computations between general pairs of program points of general while programs), and none of that work exploits summaries for termination.

Hierarchical State Machines (HSMs) [5], called Recursive State Machines (RSMs) in [2], are a model of recursive programs over finite data domains (and hence with finitely many *states*, if state refers to the valuation s of the program variables, i.e. without the stack contents γ; in our technical exposition, we use *configuration* to refer to the pair (s, γ) and avoid the term 'state' altogether).

As a side remark, we note that while loops are irrelevant in finite-state programs such as HSMs or RSMs, and can be eliminated in programs with recursion. Our exposition (for programs with while loops and recursion) permits to compare summaries for while loops with the summaries for recursive procedures replacing them.

The model checking algorithms in [5] and in [2] account for temporal properties including termination and liveness. Hence, one may wonder whether one can not prove those properties for general recursive programs by first abstracting them to finite-state recursive programs (using e.g. predicate abstraction as in [3]) and then applying those model checking algorithms. The answer is: no, one can not. Except for trivial cases, the termination or liveness property gets lost in the abstraction step. In the automation of our proof rule by abstract interpretation, one may use the idea of transition predicate abstraction [15] to obtain abstractions of summaries; a related idea, developed independently, appears in [11].

The model checking algorithms in [5] and in [2] are based on the automata-theoretic approach. In [5], the construction of a monitor Buechi automaton for the LTL or CTL* property is followed by a reachability analysis for the monitored HSM in two phases. First, summary edges from call to return of a module and path edges from entry nodes of a module to an arbitrary node in the same module are constructed. Additionally, it is indicated whether those paths pass an accepting state of the monitor. Second, the graph of a Kripke structure augmented with summary and path edges is checked for cycles. If a cycle through an accepting path exists the Buechi acceptance condition is satisfied and the property fails.

In [5], the construction of summary edges follows the fundamental graph-theoretic set-up of [19]. In [2], a (closely related) setup of Datalog rules is used. The fixpoint system that we use (in our proof rule in order to validate a summary for a given program) are reminiscent of those Datalog rules; for a rough comparison one may say that we generalize the Datalog rules from propositional to first-order logic. This is needed for

the incorporation of infinite data types, which in fact is mentioned as a problem for future work in [2].

The CaRet logic in [1] expresses properties of recursive state machines, such as non-regular properties concerning the call stack, that go beyond the properties considered in this paper (which refer to program variables only). The model checking algorithm for CaRet presented in [1] uses summary edges for procedures as in [2] and is again restricted to finite data types.

The model checker Bebob [4], a part of the SLAM model checking tool [3], is based on the construction of procedure summaries adapted from [19] using CFL-reachability. The applied algorithm is again a two stage process. First, path and summary edges are constructed and then, the actual reachability analysis is carried out by using summary and path edges. Bebop applies to C-like structured programs with procedures and recursion and no other than Boolean variables.

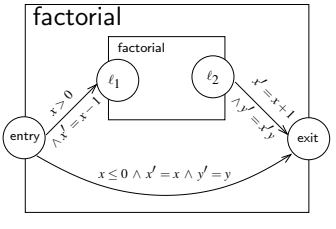
The work presented here is related to the work on program termination in [13, 14, 15] in the following way. The notion of transition invariants introduced in [14] for characterizing termination can be instantiated for recursive programs in either of two ways, by referring to program valuations (i.e. without stack contents) or by referring to configurations (i.e. pairs of program valuations and stack contents). Either case does not lead to useful proof rules for total correctness. The notion of summaries, and its putting to use for termination proofs for recursive programs, are contributions proper to this paper. The work in [14] and in [15] is relevant for the automation of our proof rule in two different ways. The algorithm presented in [13] can be used to efficiently check the third condition of the proof rule. As mentioned above, the abstraction investigated in [15] can be used to approximate summaries (and thus automate their construction by least-fixpoint iteration).

As pointed out by an anonymous referee, it is possible to define summaries using the formalism of so-called weighted pushdown systems [6, 20]. This would be useful in order to give an alternative view on our results in this framework.

3 Examples

We consider the program factorial below. We will construct a summary for the program and use the summary for proving total correctness. We hereby informally instantiate the proof rule that we will introduce in Section 6. The semantics of procedure calls is call by reference.

$$
\begin{array}{lll}
\mathsf{factorial}(x,y) = & \mathsf{entry}: & \textbf{if}\quad x>0 \\
 & & \{ \\
 & & \quad x=x-1; \\
 & \ell_1: & \quad \mathsf{factorial}(x,y); \\
 & \ell_2: & \quad x=x+1; \\
 & & \quad y=x\cdot y; \\
 & & \} \\
 & \mathsf{exit}: &
\end{array}
$$



In the abstract notation used in this paper, the program consists of one module M_0 given by a set Cmds_0 of three *commands* and a set Calls_0 of one *call*.

$$\begin{aligned}\mathsf{Cmds}_0 = \{\ & (\mathsf{entry},\ \ x \le 0 \wedge x' = x \wedge y' = y\,,\ \ \mathsf{exit})\,,\\ & (\mathsf{entry},\ x > 0 \wedge x' = x-1 \wedge y' = y\,,\ \ell_1)\,,\\ & (\ell_2,\ \ x' = x+1 \wedge y' = x'y\,,\ \ \mathsf{exit})\ \}\\ \mathsf{Calls}_0 = \{\ & (\ell_1,\ \ 0,\ \ \ell_2)\ \}\end{aligned}$$

The one-step transition relation R over program valuations is specified by the assertions $R1$ to $R5$ below. The assertions $R1$ to $R3$ correspond to the execution of the commands in $Cmds_0$ (and are obtained by their direct translation). The assertions $R4$ and $R5$ correspond to the execution of a call; we will see further below how we can obtain $R4$ and $R5$.

As usual, we express a binary relation over program valuations as a set of valuations of the program variables and the primed version of the program variables. The program variables include the program counter pc which ranges over the four locations (entry, exit, ℓ_1 and ℓ_2) of the program.

$$\begin{aligned}R1\quad & \mathsf{pc} = \mathsf{entry} \wedge x \le 0 \wedge x' = x \wedge y' = y \wedge \mathsf{pc}' = \mathsf{exit}\\ R2\quad & \mathsf{pc} = \mathsf{entry} \wedge x > 0 \wedge x' = x-1 \wedge \mathsf{pc}' = \ell_1\\ R3\quad & \mathsf{pc} = \ell_2 \wedge x' = x+1 \wedge y' = x'y \wedge \mathsf{pc}' = \mathsf{exit}\\ R4\quad & \mathsf{pc} = \ell_1 \wedge x \le 0 \wedge x' = x \wedge y' = y \wedge \mathsf{pc}' = \ell_2\\ R5\quad & \mathsf{pc} = \ell_1 \wedge x > 0 \wedge x' = x \wedge y' = (x-1)!xy \wedge \mathsf{pc}' = \ell_2\end{aligned}$$

We next consider execution sequences that contain no or *finished* recursive calls (where the final stack of the execution sequence is again the same as the initial one). The corresponding transition relation T is specified by assertions such as $T1$ and $T2$ below (we omit the other T-assertions). The assertions $T1$ and $T2$ apply to pairs of program valuations at entry and exit. The assertions $R4$ and $R5$ apply to pairs of program valuations at ℓ_1 and ℓ_2. We obtain $R4$ and $R5$ by replacing in $T1$ and $T2$ the conjuncts $\mathsf{pc} = entry$ and $\mathsf{pc}' = \mathsf{exit}$ by the conjuncts $\mathsf{pc} = \ell_1$ and $\mathsf{pc}' = \ell_2$.

$$\begin{aligned}T1\quad & \mathsf{pc} = \mathsf{entry} \wedge x \le 0 \wedge x' = x \wedge y' = y \wedge \mathsf{pc}' = \mathsf{exit}\\ T2\quad & \mathsf{pc} = \mathsf{entry} \wedge x > 0 \wedge x' = x \wedge y' = (x-1)!xy \wedge \mathsf{pc}' = \mathsf{exit}\end{aligned}$$

Finally, we consider multiple-step execution sequences with *unfinished* recursive calls (i.e. where the final stack of the execution sequence has increased by at least one item). The corresponding transition relation S is specified by assertions such as $S1$ and $S2$ below (we omit the other S-assertions).

$$\begin{aligned}S1\quad & \mathsf{pc} = \mathsf{entry}_0 \wedge x \ge 0 \wedge x' > x \wedge \mathsf{pc}' = \mathsf{entry}_0\\ S2\quad & \mathsf{pc} = \ell_1 \wedge x \ge 0 \wedge x' > x \wedge \mathsf{pc}' = \ell_1\end{aligned}$$

The disjunction of R-, S- and T-assertions (their complete list can be found in [16]) is a *summary* of the factorial program. The total correctness, specified by the pair of the precondition and the postcondition

$$\begin{aligned} \mathsf{pre} &\equiv \mathsf{pc} = \mathsf{entry} \wedge x \geq 0 \wedge y = 1 \\ \mathsf{post} &\equiv \mathsf{pc}' = \mathsf{exit} \wedge y' = x! \end{aligned}$$

follows, by the proof rule presented in Section 6, from two kinds of basic observation on the summary.

(1) The assertion $T1 \vee T2$ in conjunction with the assertion pre entails the assertion post.
(2) Each assertion denotes a well-founded relation. This is true for the assertion $S1$ by a classical argument, and it is trivially true for each of the other assertions presented here (since a relation with pairs of different locations ℓ and ℓ' admits only chains of length 1).

Second Example: Insertion Sort. In this example, reasoning over termination must account for the nesting of recursive calls and while loops. Given an array A and a positive integer n the ins_sort program sorts A. The procedure insert is applied to an array of size n and uses a while loop to insert its nth element $A[n-1]$ in its proper place, assuming that the first $n-1$ elements are sorted.

```
ins_sort(A,n) =
  entry_0 :   if n ≤ 1 then A
              else
                {
                  n = n − 1;
  ℓ_1 :           ins_sort(A,n);
  ℓ_2 :           n = n + 1;
  ℓ_3 :           insert(A,n);
                }
  exit_0 :
```

```
insert(A,n) =
  entry_1 :  i = n;
  ℓ_4 :      while (n > 1  &
                    A[n−1] < A[n−2])
               {
                 swap(A[n−2],A[n−1]);
                 n = n − 1;
               }
  ℓ_5 :      n = i;
  exit_1 :
```

A summary of the ins_sort program must account for execution sequences with nested recursion and unfolding of while loops. Again, we give a summary for the program in the form of a disjunction of R-, S- and T-assertions; see below for the ones that are most interesting for the total correctness proof.

$$\begin{aligned} T1 &\quad \mathsf{pc} = \mathsf{entry}_0 \wedge n \leq 1 \wedge \mathsf{pc}' = \mathsf{exit}_0 \\ T2 &\quad \mathsf{pc} = \mathsf{entry}_0 \wedge A'[0] \leq A'[1] \leq \ldots \leq A'[n-1] \wedge \mathsf{pc}' = \mathsf{exit}_0 \\ T3 &\quad \mathsf{pc} = \ell_4 \wedge n > 0 \wedge n' < n \wedge \mathsf{pc}' = \ell_4 \\ S1 &\quad \mathsf{pc} = \mathsf{entry}_0 \wedge n > 0 \wedge n' < n \wedge \mathsf{pc}' = \mathsf{entry}_0 \\ S2 &\quad \mathsf{pc} = \ell_1 \wedge n > 0 \wedge n' < n \wedge \mathsf{pc}' = \ell_1 \end{aligned}$$

Total correctness follows from the same two kinds of properties of the summary as in the previous example. The assertions $T1$ and $T2$ imply partial correctness if n is equal to the length of the array. Termination follows from the well-foundedness of $T3$ (which accounts for computation sequences in the while loop) and $S1$ and $S2$ (which account for the recursive descend). Note that the well-foundedness argument is itself detached from the account for (possibly nested) recursion and loops; it is applied to each assertion in isolation.

4 Recursive Programs

In this section we fix the abstract notation for general while programs with recursion. It should be straightforward to map the concrete syntax of an imperative programming language into this notation. In the remainder of the paper, we assume to have an arbitrary but fixed program $\mathcal{P}$.

- The program consists of a set of *modules* $\{M_0, \ldots, M_m\}$.
- The set of *locations* of the module M_j is denoted by Loc_j.
- Each module $\mathcal{M}_j$ has two distinguished locations noted entry_j and exit_j which are its unique *entry point* and its unique *exit point*.
- Each *command* of a module is a triple (ℓ_1, c, ℓ_2) consisting of the locations ℓ_1 and ℓ_2 of the module (the *before* and the *after* location) and the transition constraint c. A transition constraint is a formula over primed and unprimed program variables.
- Each *call* of a module is a triple (ℓ_1, k, ℓ_2) consisting of the locations ℓ_1 and ℓ_2 of the module (the *call* location and the *return* location) and the index k of the module being called (i.e. $k \in \{0, \ldots, m\}$).

The sets Cmds and Calls consist of the commands and calls, respectively, of all modules of the program. The set Loc consists of its locations, i.e. $\mathsf{Loc} = \mathsf{Loc}_0 \cup \ldots \mathsf{Loc}_m$.

The set Var consists of the program variables, which usually range over unbounded data domains. The set Var' contains the primed versions of the program variables. We use an auxiliary variable, the program counter pc, which ranges over the finite set Loc of locations of all modules.

A *program valuation* ("state") s is a valuation for the program variables and the program counter, i.e. s is a mapping from $\mathsf{Var} \cup \{\mathsf{pc}\}$ into the union of data domains. We note Σ the set of all program valuations.

A *configuration* $q = (s, \gamma)$ is a pair of a program valuation s and a word γ (the stack) over the alphabet Loc of program locations of all modules. We note Q the set of configurations; formally, $Q = \Sigma \times \mathsf{Loc}^\star$.

In assertions we use γ as a "stack variable", i.e. a variable that ranges over $\mathsf{Loc}^\star$. An assertion (e.g. a first-order formula) over the set of variables $\mathsf{Var} \cup \{\mathsf{pc}\} \cup \{\gamma\}$ denotes a set of configurations. For example, the set of initial configurations is denoted by the assertion $\mathsf{pc} = \mathsf{entry}_0 \wedge \gamma = \varepsilon$ where entry_0 is the entry location of the designated 'main' module M_0 and ε is the empty stack. An assertion over the set of variables $\mathsf{Var} \cup \{\mathsf{pc}\} \cup \{\gamma\} \cup \mathsf{Var}' \cup \{\mathsf{pc}'\} \cup \{\gamma'\}$ denotes a binary relation over configurations.

We note $\rightsquigarrow$ the *transition relation over configurations*, i.e. $\rightsquigarrow \subseteq Q \times Q$. The three different types of transitions are: local transition inside a single module, call of another module and return from a module. The transition relation $\rightsquigarrow$ is denoted by the disjunction of the assertions below.

$$\begin{array}{llll}
\mathsf{pc} = \ell_1 & \wedge\ \mathsf{pc}' = \ell_2 & \wedge\ c & \wedge\ \gamma' = \gamma \qquad \text{where } (\ell_1, c, \ell_2) \in \mathsf{Cmds} \\
\mathsf{pc} = \ell_1 & \wedge\ \mathsf{pc}' = \mathsf{entry}_j & \wedge\ \mathsf{Var}' = \mathsf{Var} & \wedge\ \gamma' = \ell_2.\gamma \qquad \text{where } (\ell_1, j, \ell_2) \in \mathsf{Calls} \\
\mathsf{pc} = \mathsf{exit}_j & \wedge\ \mathsf{pc}' = \ell_2 & \wedge\ \mathsf{Var}' = \mathsf{Var} & \wedge\ \gamma = \ell_2.\gamma' \qquad \text{where } (\ell_1, j, \ell_2) \in \mathsf{Calls}
\end{array}$$

According to the three kinds of assertions, we distinguish three kinds of transitions.

A *local transition* $q \rightsquigarrow q'$ is induced by a command (ℓ_1, c, ℓ_2) of the module. It is enabled in the configuration q if the values of the program variables satisfy the guard

formula in the transition constraint c of the command at the corresponding location ℓ_1. The program counter and the program variables are updated in q' accordingly; the stack remains unchanged.

Both, a *call* and a *return transition* $q \rightsquigarrow q'$, are induced by a call command (ℓ_1, j, ℓ_2) calling a module M_j. In both, the stack γ is updated and the program variables remain unchanged ($\mathsf{Var}' = \mathsf{Var}$ stands for the conjunction of $x' = x$ over all program variables x).

In a *call* transition the stack is increased by the return location ℓ_2 (by a *push* operation). The value of the program counter is updated to the entry location entry_j of the module M_j being called.

When the exit location of the called module M_j is reached, the control flow returns to the return location ℓ_2 of the calling module, which is the top value of the return stack. Thus, in a *return* transition, the value of the program counter is updated by the top value of the stack, and the stack is updated by removing its top element (by a *pop* operation).

A (possibly infinite) *computation* is a sequence of configurations $q_0, q_1, q_2, \ldots$ that starts with an initial configuration and that is consecutive, i.e. $q_i \rightsquigarrow q_{i+1}$ for all $i \geq 0$.

5 Summaries

In its generalized form that we introduce in this section, a summary captures the effect of computations that start and end at any pair of program points (and not just to the pair of the entry and exit points of a module). The computations in questions may contain calls that are not yet returned; i.e., in general they don't obey to the '*each call is matched by a subsequent return*' discipline. We first introduce the corresponding *transition relation over program valuations* the *descends* relation, noted $\xrightarrow{\leq}$.

Definition 1 (Intraleads ($\xrightarrow{=}$), Strictly Descends ($\xrightarrow{<}$), Descends ($\xrightarrow{\leq}$)). The pair (s, s') of program valuations lies in the *intraleads* relation if a configuration (s, γ) can go to the configuration (s', γ) (with the same stack) via a *local* transition or via the *finished* execution of a call statement.

$$s \xrightarrow{=} s' \text{ if } (s,\gamma) \rightsquigarrow (s',\gamma) \text{ or } (s,\gamma) \rightsquigarrow (s_1,\ell.\gamma) \rightsquigarrow (s_2,\gamma_2) \rightsquigarrow \ldots \rightsquigarrow (s_{n-1},\gamma_{n-1}) \ldots \rightsquigarrow (s_n,\ell.\gamma) \rightsquigarrow (s',\gamma)$$
$$\text{where } \gamma \in \mathsf{Loc}^\star,\ \ell \in \mathsf{Loc}, \text{ and } \gamma_2, \ldots, \gamma_{n-1} \text{ contain } \ell.\gamma \text{ as suffix}$$

The pair (s, s') of program valuations lies in the *strictly descends* relation if a configuration (s, γ) can go to a configuration $(s', \ell.\gamma)$ via a *call* transition.

$$s \xrightarrow{<} s' \text{ if } (s,\gamma) \rightsquigarrow (s',\ell.\gamma) \quad \text{where } \gamma \in \mathsf{Loc}^\star \text{ and } \ell \in \mathsf{Loc}$$

The *descends* relation $\xrightarrow{\leq}$ is the union of the two relations above.

$$\xrightarrow{\leq} \;=\; \xrightarrow{=} \cup \xrightarrow{<}$$

We can now define summaries.

Definition 2 (Summary). A summary $\mathcal{S}$ is a binary relation over program valuations that contains the transitive closure of its descends relation.

$$\mathcal{S} \quad \supseteq \quad \xrightarrow{\leq}^{+}$$

In other words, a summary $\mathcal{S}$ contains a pair (s,s') of program valuations if there exists a computation from a configuration (s,γ) to a configuration (s',γ') such that the initial stack γ is a suffix not only of the final stack γ' but also of every intermediate stack.

Summaries as Fixpoints. The fixpoint system below[1] is a conjunction of inclusions between relations over valuations.

Fixpoint System Φ(R,S,T)

$I1$	R	$\supseteq$	$(\mathsf{pc} = \ell_1 \wedge c \wedge \mathsf{pc}' = \ell_2)$	$(\ell_1, c, \ell_2) \in \mathsf{Cmds}$
$I2$	T	$\supseteq$	$R \cup T \circ R$	
$I3$	R	$\supseteq$	$(\mathsf{pc} = \ell_1 \wedge c \wedge \mathsf{pc}' = \ell_2)$ if	
	T	$\supseteq$	$(\mathsf{pc} = \mathsf{entry}_j \wedge c \wedge \mathsf{pc}' = \mathsf{exit}_j)$	$(\ell_1, j, \ell_2) \in \mathsf{Calls}$
$I4$	S	$\supseteq$	$(\mathsf{pc} = \ell_1 \wedge \mathsf{Var}' = \mathsf{Var} \wedge \mathsf{pc}' = \mathsf{entry}_j)$	$(\ell_1, j, \ell_2) \in \mathsf{Calls}$
$I5$	S	$\supseteq$	$S \circ (\mathsf{pc} = \ell_1 \wedge \mathsf{Var}' = \mathsf{Var} \wedge \mathsf{pc}' = \mathsf{entry}_j)$	$(\ell_1, j, \ell_2) \in \mathsf{Calls}$
$I6$	S	$\supseteq$	$S \circ T \cup T \circ S$	

A fixpoint is a triple (R,S,T) that satisfies all inclusions of the form $I1$ to $I6$. It can be computed by least fixpoint iteration of (an abstraction of) the operator defined by the fixpoint system. The operator induced by $I3$ takes a set of pairs of valuations, restricts it to pairs at entry and exit locations and replaces them with the corresponding pairs at call and return locations.

Theorem 1. *If the three relations over program valuations R, S and T form a fixpoint for the fixpoint system Φ, their union $\mathcal{S} = R \cup T \cup S$ is a summary for the program.*

The theorem follows from Lemmas 1 and 2 below.

Lemma 1. *The relation T is a superset of the transitive closure of the intraleads relation.*

$$T \quad \supseteq \quad \xrightarrow{=}^{+} \tag{1}$$

[1] In our notation, we identify an assertion with the relation that it denotes. We use the operator $\circ$ for relational composition. That is, for binary relations A and B,

$$A \circ B = \{(s,s'') \mid \exists s' : (s,s') \in A \wedge (s',s'') \in B\}.$$

Proof. It is sufficient to show the statement below, which refers to configurations whose stack is empty.

If (s',ε) is $\rightsquigarrow$ -reachable from (s,ε), then T contains (s,s').

We proceed by induction over the computation that leads from (s,ε) to (s',ε).

Base Step $(s,\varepsilon) \rightsquigarrow (s',\varepsilon)$.
The only one-step transition that does not change the stack is a *local* transition, i.e. the valuation (s,s') satisfies an assertion of the form $\mathsf{pc} = \ell_1 \wedge \mathsf{pc}' = \ell_2 \wedge c$ where (ℓ_1, c, ℓ_2) is a command in Cmds. By inclusions $I1$ and $I2$, R and thus also T contains (s,s').

Induction Step $(s,\varepsilon) \rightsquigarrow (s_1,\gamma_1) \rightsquigarrow \ldots \rightsquigarrow (s_n,\gamma_n) \rightsquigarrow (s',\varepsilon)$**.**

Case 1. The computation from (s,ε) to (s',ε) contains no intermediate configuration with empty stack.
The stack γ_1 of the second configuration consists of one location ℓ_1, i.e. $\gamma_1 = \ell_1$, and it is equal to the stack γ_n of the last but one configuration.
The transition $(s,\varepsilon) \rightsquigarrow (s_1,\ell_1)$ is a call transition induced by, say, the call (ℓ_1,k,ℓ_2). This means that the value of the program counter in s_1 is the entry location entry_k of the called module $\mathcal{M}_k$.
The transition $(s_n,\ell_1) \rightsquigarrow (s',\varepsilon)$ is a return transition. This means that the value of the program counter in s_n is the exit location exit_k of the called module $\mathcal{M}_k$.
The computation from (s_1,ℓ_1) to (s_n,ℓ_1) is an execution (in $\mathcal{M}_k$) from entry_k to exit_k. Since no intermediate configuration has an empty stack, every intermediate stack has ℓ_1 as its first element. Hence (s_n,ε) is $\rightsquigarrow$ -reachable from (s_1,ε). By induction hypothesis, T contains the pair (s_1,s_n). By inclusions $I2$ and $I3$, R and thus also T contain (s,s').

Case 2. The computation from (s,ε) to (s',ε) contains at least one intermediate configuration with empty stack.
We consider the subsequence of all configurations with empty stack in the computation.

$$(s,\varepsilon) \rightsquigarrow^+ (s_{i_1},\varepsilon) \rightsquigarrow^+ \ldots \rightsquigarrow^+ (s_{i_m},\varepsilon) \rightsquigarrow^+ (s',\varepsilon)$$

For each part of the computation from (s_{i_j},ε) to $(s_{i_{j+1}},\varepsilon)$, we can apply the first case (none of the intermediate configurations has an empty stack) and obtain that R contains all pairs of valuations in consecutive configurations of the subsequence. By inclusion $I2$, T is the transitive closure of R and thus contains (s,s').

□

The proof of Lemma 1 exhibits that R is a superset of the intraleads relation.

$$R \quad \supseteq \quad \xrightarrow{=} \tag{2}$$

Since $T \supseteq R^+$ holds by $I2$, inclusion (1) is a direct consequence of inclusion (2). It seems, however, impossible to show (2) without showing (1).

Lemma 2. *The relation S is a superset of the transitive closure of the descends relation minus the transitive closure of the intraleads relation.*

$$S \supseteq \xrightarrow{\leq}^{+} \setminus \xrightarrow{=}^{+}$$

Proof. Since

$$\xrightarrow{\leq}^{+} \setminus \xrightarrow{=}^{+} = (\xrightarrow{=}^{\star} \circ \xrightarrow{<} \circ \xrightarrow{=}^{\star})^{+}$$

it is sufficient to show the statement below, which refers to configurations whose stack is empty.

If (s',γ') with non-empty stack γ' is $\rightsquigarrow$ -reachable from (s,ε), then S contains (s,s').

We proceed by induction over the size d of γ'.

Base Step ($d = 1$). The computation leading from (s,ε) to (s',γ') is of the form

$$(s,\varepsilon) \rightsquigarrow^{*} (s_1,\varepsilon) \rightsquigarrow (s_2,\ell) \rightsquigarrow^{*} (s',\ell).$$

The transition $(s_1,\varepsilon) \rightsquigarrow (s_2,\ell)$ is a call transition. By inclusion $I4$, S contains (s_1,s_2). If s is different from s_1 or s' is different from s_2: by Lemma 1, T contains (s,s_1) resp. (s_2,s'), and by inclusion $I6$, S contains (s,s').

Induction Step ($d \Rightarrow d+1$). The computation is of the form

$$(s,\varepsilon) \rightsquigarrow^{+} (s_k,\gamma_k) \rightsquigarrow (s_{k+1},\ell.\gamma_k) \rightsquigarrow^{*} (s',\ell.\gamma_k).$$

By induction hypothesis, S contains (s,s_k). The transition from (s_k,γ_k) to $(s_{k+1},\ell.\gamma_k)$ is a call transition. By inclusion $I5$ of the fixpoint system, S contains (s_1,s_{k+1}). If s_{k+1} is different from s': by Lemma 1, T contains (s_{k+1},s), and by inclusion $I6$, S contains (s,s').

□

6 Total Correctness

We assume that the correctness of the program is specified by the pair of pre- and postconditions pre and post where pre is an assertion over the set Var of unprimed program variables and post is an assertion over the set $\mathsf{Var} \cup \mathsf{Var}'$ of primed and unprimed program variables. The assertions are associated with the entry and exit points of the 'main' module M_0.

Partial correctness is the following property: if a computation starts in a configuration $q = (s,\varepsilon)$ with the empty stack and the valuation s satisfying the assertion $\mathsf{pc} = \mathsf{entry}_0 \wedge \mathsf{pre}$ and terminates in a configuration $q' = (s',\varepsilon)$ with the empty stack and the valuation s' satisfying the assertion $\mathsf{pc} = \mathsf{entry}_0$, then the pair of valuations (s,s') satisfies the assertion post.

Theorem 2. *The program is partially correct if and only if there exists a summary $\mathcal{S}$ whose restriction to the precondition and the entry and exit points of the 'main' module M_0 entails the postcondition.*

$$\mathcal{S} \wedge \mathsf{pre} \wedge \mathsf{pc} = \mathsf{entry}_0 \wedge \mathsf{pc}' = \mathsf{exit}_0 \models \mathsf{post}$$

In the formulation above, the only-if direction of the theorem requires an assumption on the program syntax, namely that the 'main' module M_0 does not get called, i.e. no call is of the form $(\ell_1, 0, \ell_2)$. The assumption can always be made fulfilled by a small syntactic transformation of the program.

To see why the assumption is needed, consider the example program factorial which, in the syntax given in Section 3, does not satisfy the assumption. The S-assertion $S2$ (which refers to the precondition and the entry and exit points of the 'main' module M_0) does *not* entail the postcondition $y' = x!$ and neither does the refinement of $S2$ of the form

$$\exists n > 0 : \ \mathsf{pc} = \mathsf{entry}_0 \wedge x > 0 \wedge x' = x - n \wedge y' = (x-n)!\, y \wedge \mathsf{pc}' = \mathsf{exit}_0$$

which is contained in every summary of the program.

The assumption on the program syntax is not required in the formulation of the corollary below, which refers to the relation T.

Corollary 1. *The program is partially correct if and only if there exists a relation T over program valuations that is a solution in the fixpoint system Φ and whose restriction of T to the precondition and the entry and exit points of the 'main' module entails the postcondition.*

$$T \wedge \mathsf{pre} \wedge \mathsf{pc} = \mathsf{entry}_0 \wedge \mathsf{pc}' = \mathsf{exit}_0 \models \mathsf{post}$$

Obviously only the inclusions of the form $I1 - I3$ of Φ are relevant for a solution for T.

Termination is the property that every computation of the program, i.e. every sequence of configurations $q_0 \rightsquigarrow q_1 \rightsquigarrow q_2 \ldots$ is finite. The next theorem states that one can characterize termination in terms of summaries.

Theorem 3. *The program is terminating if and only if there exists a summary $\mathcal{S}$ that is a finite union of well-founded relations.*

Proof (Sketch). For a proof by contradiction, we assume that there exists an infinite computation $(s_0, \varepsilon), (s_1, \gamma_1), (s_2, \gamma_2), \ldots$ starting in the empty stack. We now construct an infinite subsequence of configurations $(s^0, \gamma^0), (s^1, \gamma^1), (s^2, \gamma^2), \ldots$ such that the corresponding valuations form a descending sequence.

$$s^0 \xrightarrow{\leq} s^1 \xrightarrow{\leq} s^2 \xrightarrow{\leq} \ldots$$

The first part of the subsequence of configurations consists of all configurations with an empty stack, i.e. $(s^k, \gamma^k) = (s_{i_k}, \varepsilon)$. If there are infinitely many configurations with empty stacks, then we are done with the construction and we obtain an infinite intraleads sequence.

Otherwise, there is a configuration (s_{i_k}, ε) such that the stack of all subsequent configurations is not empty.

The transition from (s_{i_k}, ε) to (s_{i_k+1}, ℓ) is a call transition. Hence the pair of valuations (s_{i_k}, s_{i_k+1}) is in $\xrightarrow{<}$.

We repeat the above construction step with (s_{i_k+1}, ℓ) instead of (s_0, ε). Inductively we get an infinite sequence $s^0, s^1, s^2, \ldots$ of valuations such that pairs of consecutive valuations are in $\xrightarrow{\leq}$ and hence in $\mathcal{S}$.

We now use the assumption that $\mathcal{S}$ is a finite union of well-founded relations, say[2]

$$\mathcal{S} = \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_m.$$

We define a function f with finite range that maps an ordered pair of indices of elements of the sequence $s^0, s^1, s^2 \ldots$ to the index j of the relation $\mathcal{S}_j$ that contains the corresponding pair of valuations.

$$f(k,l) \stackrel{\text{def.}}{=} j \quad \text{where } (s^k, s^l) \in \mathcal{S}_j$$

The function f induces an equivalence relation $\sim$ on pairs of indices of $s^0, s^1, s^2, \ldots$.

$$(k_1, l_1) \sim (k_2, l_2) \quad \stackrel{\text{def.}}{\Leftrightarrow} \quad f(k_1, l_1) = f(k_2, l_2).$$

The index of $\sim$ is finite since the range of f is finite. By Ramsey's theorem [18], there exists an infinite set of indices K such that all pairs from K belong to the same equivalence class. Thus, there exists m and n in K, with $m < n$, such that for every k and l in K, with $k < l$, we have $(k,l) \sim (m,n)$. Let $k_1, k_2, \ldots$ be the ascending sequence of elements of K. Hence, for the infinite sequence $s^{k_1}, s^{k_2}, \ldots$ we have $(s^{k_1}, s^{k_i}) \in \mathcal{S}_j$ for all $i \geq 1$. But this is a contradiction to the fact that $\mathcal{S}_j$ is well-founded.

□

Corollary 2. *The program is terminating if and only if there exist three relations over program valuations R, S and T that form a solution of the of the fixpoint system Φ and that are finite unions of well-founded relations.*

Deductive Verification. Below we give a proof rule for the total correctness of general while programs with recursion. The proof rule is sound and complete by Theorem 1 and Corollaries 1 and 2.

Deductive verification according to the proof rule proceeds in three steps, for three given relations R, S and T over program valuations. The first step checks that the triple (R,S,T) is a fixpoint, i.e. that the relations R, S and T satisfy the inclusions given under $I1 - I6$ of the fixpoint system of Section 5. The second step checks that the restriction of the relation T to the precondition and the entry and exit points of the 'main' module entails the postcondition. The third step checks that $R \cup S \cup T$ is a finite union of well-founded relations.

[2] The assumption implies that one of the relations $\mathcal{S}_j$ occurs infinitely often in the sequence $s^0, s^1, s^2, \ldots$. This is, however, not yet a contradiction to the well-foundedness of $\mathcal{S}_j$, which needs a consecutive $\mathcal{S}_j$ -sequence.

$\mathcal{P}$: program
R, T, S : assertions over pairs of valuations
$\mathsf{pre}, \mathsf{post}$: pre- and postconditions for $\mathcal{P}$

1. R, S and T form a fixpoint of Φ.
2. $T \wedge \mathsf{pre} \wedge \mathsf{pc} = \mathsf{entry}_0 \wedge \mathsf{pc}' = \mathsf{exit}_0 \models \mathsf{post}$
3. T and S are finite unions of well-founded relations.

Total correctness of $\mathcal{P}$: $\{\mathsf{pre}\}\ \mathcal{P}\ \{\mathsf{post}\}$

An informal description of an application of the above proof rule has been given in Section 3. It is now straightforward to instantiate the proof rule also formally for the presented examples.

Automatic Verification. The inclusions $I1 - I6$ of the fixpoint system and the condition for partial correctness amounts to checking entailment between assertions. Checking the well-foundedness of the finitely many member-relations of S and T can be established automatically in many cases; see [13, 22, 12, 7]. The synthesis of the relations R, S and T is possible by least fixpoint iteration (over the domain of relations over program valuations) in combination with abstract interpretation methods [8, 9].

7 Conclusion

We have introduced a generalization of the fundamental notion of procedure summaries. Our summaries refer to arbitrarily precise descriptions of the effect of computations between general pairs of program points of general while programs (over in general infinite data domains). We have shown how one can put them to work for the verification of termination and total correctness of general while programs with recursion.

We have presented a proof rule for total correctness that uses summaries as the auxiliary assertion on the program. As already mentioned, the proof rule has an obvious potential for automation via abstract interpretation. We believe that our paper may lead to several directions of follow-up work to realize this potential, with a choice of abstraction methods (see e.g. [8, 9, 15, 11]) and techniques for the efficient construction of summaries (see e.g. [19, 2]). Other lines of future work are the extension to concurrent threads (see e.g. [10, 17]) and the account of correctness properties expressed in the CaRet logic [1].

Acknowledgements. We thank Alexander Malkis and Andrey Rybalchenko for comments and suggestions. We acknowledge partial support by the German Research Council (DFG) through SFB/TR 14 AVACS and by the German Federal Ministry of Education, Science, Research and Technology (BMBF) through the Verisoft project.

References

1. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Proceedings of TACAS'04*, 2004.
2. R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proceedings of CAV'00*, 2000.
3. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of PLDI'2001*, 2001.
4. T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. *In SPIN Workshop on Model Checking of Software*, 2000.
5. M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestrcited hierachical state machines. In *Proceedings of ICALP 2001*, 2001.
6. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *In Proceedings of POPL'03*.
7. M. Colón and H. Sipma. Synthesis of linear ranking functions. In *Proceedings of TACAS'01*, 2001.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Proceedings of POPL'77*, 1977.
9. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of POPL'1979*, 1979.
10. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN03: SPIN Workshop*. Spiegel-Verlag, 2003.
11. B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *In Proceedings of SAS'04*, 2004.
12. D. A. McAllester and K. Arkoudas. Walther recursion. In *CADE'96*, 1996.
13. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Proceedings of VMCAI'04*, 2004.
14. A. Podelski and A. Rybalchenko. Transition invariants. In *Proceedings of LICS'04*, 2004.
15. A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *Proceedings of POPL'05*, 2005.
16. A. Podelski, I. Schaefer, and S. Wagner. Summaries for While Programs with Recursion. Technical Report MPI-I-2004-1-007, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2004.
17. S. Qadeer, S.Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Proceedings of POPL'04*, 2004.
18. F. P. Ramsey. On a problem of formal logic. In *Proceedings London Math. Soc.*, 1930.
19. T. Reps, M. Sagiv, and S. Horwitz. Precise interprocedural dataflow analysis via graph reachability. *Proceedings of POPL'95*, 1995.
20. T.W. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Proceedings of SAS'03*, 2003.
21. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, 1981.
22. A. Tiwari. Termination of linear programs. In *Proceedings of CAV'04*, 2004.
23. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of LICS'86*, 1986.

Determinacy Inference for Logic Programs

Lunjin Lu and Andy King

[1] Department of Computer Science, Oakland University,
Rochester, MI 48309, USA
[2] Computing Laboratory, University of Kent,
Canterbury, CT2 7NF, UK

Abstract. This paper presents a determinacy inference analysis for logic programs. The analysis infers determinacy conditions that, if satisfied by a call, ensures that it computes at most one answer and that answer is generated only once. The principal component of the technique is a goal-independent analysis of individual clauses. This derives a condition for a call that ensures only one clause in the matching predicate possesses a successful derivation. Another key component of the analysis is backwards reasoning stage that strengthens these conditions to derive properties on a call that assure determinacy. The analysis has applications in program development, implementation and specialisation.

1 Introduction

One issue in logic programming is checking whether a given program and goal are deterministic [3, 5, 7, 11, 14], that is, whether the goal has at most one computed answer (and whether that answer is generated only once). As well as being key to efficient parallel [6, 10] and sequential [5, 7] implementations, knowledge of determinacy is important in program development and O'Keefe [18] writes,

> "you should keep in mind which ... predicates are determinate, and when they are determinate, and you should provide comments for your own code to remind you of when your own code is determinate."

This paper presents a determinacy inference analysis. It synthesises a determinacy condition for each predicate that, if satisfied by a call to the predicate, guarantees that there is at most one computed answer for the call and that the answer is produced only once if ever. Determinacy inference generalises determinacy checking; rather than verify that a particular goal is deterministic it deduces, in a *single* application of the analysis, a class of goals that are deterministic. The analysis has the advantage of being goal-independent, and is a step towards the ideal when human interaction is only required to inspect the answers. More exactly, the analysis can be activated by directing the analyser to a source file and pressing a button; in contrast to goal-dependent techniques the analysis does not require a top-level goal or module entry point to be specified by the programmer. As well as being applicable to determinacy checking problems,

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 108–123, 2005.

the analysis opens up new applications. In program development, if the class of deterministic goals inferred by the analysis does not match the expectations of the programmer, then the program is possibly buggy, or at least inefficient. In program specialisation, it is well-known that if a non-leftmost call in a goal is unfolded, then the corresponding residual program can be less efficient than the original one. Determinacy inference can address this issue and it seems promising as a form of binding-time analysis in the so-called semi-online (or mixline) approach to program specialisation [13]. In this scheme, the usual static and dynamic polarisation that is used within classic binding-time analysis [4] is refined by adding an additional binding-type semi. As well as always unfolding a static call and never unfolding a dynamic call, the unfolding decision for a semi call is postponed until specialisation time. Determinacy inference fits into this scheme because a goal-dependent binding-time analysis can verify whether a determinacy condition for a given call is satisfied and therefore whether the call is determinate. These calls can then be marked as semi since determinate goals are prime candidates for unfolding. These semi calls are also annotated with lightweight conditions that select the clause with which to unfold the call. The net result is more aggressive unfolding. To summarise, this paper makes the following contributions:

- it presents a determinacy inference technique which generalises previously proposed determinacy checking techniques. The analysis also has applications in the burgeoning area of semi-online program specialisation;
- it shows how determinacy inference can be decomposed into the sub-problems of (1) deriving a mutual exclusion condition on each call that ensures that only one matching clause has a successful derivation; (2) applying backward reasoning to enrich these conditions on calls so as to assure determinacy;
- it shows that (1) can be tackled with techniques such as depth-k [20] and argument-size analysis [2] when suitably augmented with a projection step; and that (2) can be tackled with backward analysis [12];
- it reports experimental work that provides evidence that the method scales and that it can infer rich (and sometimes surprising) determinacy conditions.

Section 2 illustrates the key ideas with a worked example. Section 3 explains how mutual exclusion conditions can be derived that, when satisfied by a call, ensures that no more than one clause of the matching predicate can lead to a successful derivation. Section 4 presents a backward analysis that strengthens these conditions to obtain determinacy conditions. Section 5 details an initial experimental evaluation and sections 6 and 7 the related work and conclusions. To make the ideas accessible to a wider programming language audience, the analysis is, wherever possible, presented informally with minimal notation.

2 Worked Example

This section explains the analysis by way of an example. In order to derive conditions on calls that are sufficient for determinacy, it is necessary to reason

about individual success patterns of the constituent clauses of a predicate. In particular, it is necessary to infer conditions under which the success patterns for any pair of clauses do not overlap. This can be achieved by describing success patterns with suitable abstractions. One such abstraction can be constructed from the list-length norm that is defined as follows:

$$\|t\| = \begin{cases} t & \text{if } t \text{ is a variable} \\ 1 + \|t_2\| & \text{if } t = [t_1|t_2] \\ 0 & \text{otherwise} \end{cases}$$

The norm maps a term to a size that is in fact a linear expression defined over the natural numbers and the variables occurring in the term. Observe that if two terms t_1 and t_2 are described by constant expressions of different value, that is $\|t_1\| \neq \|t_2\|$, then t_1 and t_2 are distinct. In fact, to reason about non-overlapping sets of success patterns (rather than sets of terms), it is necessary to work with argument-size relationships [2] which are induced from a given norm. To illustrate argument-size relationships, and their value in determinacy inference, the following program will be used as a running example throughout this section. The program, like all those considered in the paper, is flat in the sense that the arguments of atoms are vectors of distinct variables. Clauses are numbered for future reference.

```
(1) append(Xs, Ys, Zs) :-
        Xs = [], Ys = Zs.
(2) append(Xs, Ys, Zs) :-
        Xs = [X|Xs1],
        Zs = [X|Zs1],
        append(Xs1, Ys, Zs1).
```

```
(3) rev(Xs,Ys) :-
        Xs = [], Ys = [].
(4) rev(Xs,Ys) :-
        Xs = [X|Xs1], Ys2 = [X],
        rev(Xs1, Ys1),
        append(Ys1, Ys2, Ys).
```

2.1 Computing Success Patterns

To derive size relationships the program is abstracted by applying the norm to the terms occurring within it. Applying the norm to the terms in a syntactic equation $t_1 = t_2$ yields a linear equation $\|t_1\| = \|t_2\|$. The key idea is that a variable in the derived program – the so-called abstract program – describes the size of the corresponding variable in the original program. Since term sizes are non-negative, it is safe to additionally assert that each variable in the abstract program is non-negative. The abstract program thus obtained is listed below.

```
(1) append(Xs, Ys, Zs) :-
        Xs ≥ 0,Ys ≥ 0,Zs ≥ 0,
        Xs = 0, Ys = Zs.
(2) append(Xs, Ys, Zs) :-
        Xs ≥ 0,Ys ≥ 0,Zs ≥ 0,
        Xs1 ≥ 0,Zs1 ≥ 0,
        Xs = 1 + Xs1,
        Zs = 1 + Zs1,
        append(Xs1, Ys, Zs1).
```

```
(3) rev(Xs, Ys) :-
        Xs ≥ 0,Ys ≥ 0,
        Xs = 0, Ys = 0.
(4) rev(Xs,Ys) :-
        Xs ≥ 0,Ys ≥ 0,
        Xs1 ≥ 0,Ys1 ≥ 0,Ys2 ≥ 0,
        Xs = 1 + Xs1, Ys2 = 1,
        rev(Xs1,Ys1),
        append(Ys1,Ys2,Ys).
```

The value of the abstract program is that its success patterns, which are given below, describe size attributes of the original program. The key idea is that the success sets of the abstract program faithfully describe the size relationships on the success sets of the original program.

$$\texttt{append}(x_1, x_2, x_3) \text{ :- } (x_1 \geq 0) \wedge (x_2 \geq 0) \wedge (x_1 + x_2 = x_3)$$
$$\texttt{rev}(x_1, x_2) \text{ :- } (x_1 \geq 0) \wedge (x_1 = x_2)$$

The relation $x_1 + x_2 = x_3$ captures the property that if the original program is called with a goal $\texttt{append}(x_1, x_2, x_3)$ then any computed answer will satisfy the property that the size of x_1 wrt $\|.\|$, when summed with the size of x_2 will exactly coincide with the size of x_3. Moreover, the success patterns that are systems of linear inequalities can be inferred automatically by mimicking the T_P operator [21] and specifically calculating a least fixpoint (lfp) [2].

2.2 Synthesizing Mutual Exclusion Conditions

Mutual exclusion conditions are synthesized next; one condition for each predicate in the program. Such a condition, if satisfied by a call, guarantees that if one clause of the predicate can lead to a solution then no other clauses can do so. For example, one mutual exclusion condition for `append` is that its first argument is bound to a non-variable term. If the first clause leads to a solution, then its head must unify with the call. Thus the second clause cannot match the call and *vice versa*. Notice, that mutual exclusion is not sufficient for determinacy. For instance, the call `append([W|X],Y,Z)` – which satisfies the above mutual exclusion condition – possesses multiple solutions. Mutual exclusion conditions are synthesised by computing success patterns for individual clauses. This is accomplished by evaluating the body of an abstract clause with the predicate-level success patterns. This yields the following clause-level success patterns:

$$\begin{array}{ll}
1 & \texttt{append}(x_1, x_2, x_3) \text{ :- } (x_1 = 0) \wedge (x_2 \geq 0) \wedge (x_2 = x_3) \\
2 & \texttt{append}(x_1, x_2, x_3) \text{ :- } (x_1 \geq 1) \wedge (x_2 \geq 0) \wedge (x_1 + x_2 = x_3) \\
3 & \texttt{rev}(x_1, x_2) \text{ :- } (x_1 = 0) \wedge (x_2 = 0) \\
4 & \texttt{rev}(x_1, x_2) \text{ :- } (x_1 \geq 1) \wedge (x_1 = x_2)
\end{array}$$

The next step is to compute a rigidity property that guarantees that at most one of its clauses can yield a computed answer. A term t is rigid wrt to a norm $\|.\|$ if $\|t\|$ is a fixed constant. For example, a term t is rigid wrt list-length if t is *not* an open list. More generally, a Boolean function such as $x_1 \wedge (x_2 \leftrightarrow x_3)$ can express rigidity constraints on the arguments of a call; it states that x_1 is rigid wrt $\|.\|$ and that x_2 is rigid iff x_3 is rigid. Suppose now that $p(\boldsymbol{x})$:-c_1 and $p(\boldsymbol{x})$:-c_2 are success patterns for two clauses. A rigidity constraint on the arguments of $p(\boldsymbol{x})$ that is sufficient for mutual exclusion can be computed by:

$$\mathcal{X}_P(p(\boldsymbol{x})) = \bigvee \{\wedge Y \mid Y \subseteq var(\boldsymbol{x}) \wedge (\overline{\exists} Y(c_1) \wedge \overline{\exists} Y(c_2) = false)\}$$

where $var(o)$ is the set of variables occurring in the syntactic object o. The projection operator $\overline{\exists} Y(c)$ maps c onto a weaker linear constraint that ranges

over variables in the set Y. For example, $\overline{\exists}\{x_2\}((x_1 \geq 1) \wedge (x_1 = x_2)) = (x_2 \geq 1)$. If $\overline{\exists}Y(c_1) \wedge \overline{\exists}Y(c_2)$ is unsatisfiable, then the Boolean formula $\wedge Y$ expresses a rigidity condition on the arguments of $p(\boldsymbol{x})$ that is sufficient for mutual exclusion. To see this, observe that if the arguments in Y are rigid, then their sizes cannot change as execution proceeds. Thus the projection $\overline{\exists}Y(c_i)$ holds at the selection of the respective clause since it holds at the end of a successful derivation. Since $\overline{\exists}Y(c_1) \wedge \overline{\exists}Y(c_2)$ is unsatisfiable when Y are rigid, $\wedge Y$ is enough for mutual exclusion. This tactic generates the following conditions for the reverse program which states that the clauses of $\texttt{append}(x_1, x_2, x_3)$ are mutually exclusive if either x_1 is rigid or both x_2 and x_3 are rigid.

$$\mathcal{X}_P(\texttt{append}(x_1, x_2, x_3)) = \vee\{\wedge\{x_1\}, \wedge\{x_2, x_3\}, \wedge\{x_1, x_2, x_3\}\} = x_1 \vee (x_2 \wedge x_3)$$
$$\mathcal{X}_P(\texttt{rev}(x_1, x_2)) = \vee\{\wedge\{x_1\}, \wedge\{x_2\}\} = x_1 \vee x_2.$$

2.3 Synthesizing Determinacy Conditions

The last phase in determinacy inference involves calculating a rigidity constraint for each predicate such that any call satisfying the constraint will yield at most one computed answer. This is achieved with backward analysis [12] which computes a greatest fixpoint (gfp) to strengthen the mutual exclusion conditions into determinacy conditions. The gfp makes use of rigidity success patterns which are computed, again, by simulating the T_P operator in a lfp calculation.

Least Fixpoint. The lfp calculation is performed on another abstract version of the program. This version is obtained by replacing syntactic constraints with rigidity constraints. For example, `Xs = 1 + Xs1` is replaced with the Boolean formula $\texttt{Xs} \leftrightarrow \texttt{Xs1}$ which expresses that `Xs` is rigid wrt list-length iff `Xs1` is rigid. The abstract program obtained in this fashion is given below.

```
(1) append(Xs, Ys, Zs) :-
        Xs,Ys ↔ Zs.
(2) append(Xs, Ys, Zs) :-
        Xs ↔ Xs1,
        Zs ↔ Zs1,
        append(Xs1, Ys, Zs1).
(3) rev(Xs, Ys) :-
        Xs, Ys.
(4) rev(Xs,Ys) :-
        Xs ↔ Xs1, Ys2,
        rev(Xs1, Ys1),
        append(Ys1, Ys2, Ys).
```

Interpreting this program with a version of T_P that operates over Boolean functions, the following success patterns are obtained that express rigidity properties of the original program. The pattern for `rev`, for instance, states x_1 is rigid iff x_2 is rigid in any computed answer to $\texttt{rev}(x_1, x_2)$ in the original program.

$$\texttt{append}(x_1, x_2, x_3) \text{ :- } x_1 \wedge (x_2 \leftrightarrow x_3) \qquad \texttt{rev}(x_1, x_2) \text{ :- } x_1 \wedge x_2$$

Greatest Fixpoint. Each iteration in the gfp calculation amounts to:

- deriving a determinacy condition for each clause of the predicate that ensures no more than one derivation commencing with that clause may succeed;
- conjoining these conditions with the mutual exclusion of the predicate.

The single resulting condition, which is expressed as rigidity constraint, defines the next iterate. The iterates that arise when processing reverse are given below:

$$I_0 = \left\{ \begin{array}{r} \texttt{append}(x_1, x_2, x_2) \text{ :- } \mathit{true} \\ \texttt{rev}(x_1, x_2) \text{ :- } \mathit{true} \end{array} \right\} \quad I_1 = \left\{ \begin{array}{r} \texttt{append}(x_1, x_2, x_2) \text{ :- } x_1 \vee (x_2 \wedge x_3) \\ \texttt{rev}(x_1, x_2) \text{ :- } x_1 \vee x_2 \end{array} \right\}$$

$$I_3 = I_2 = \left\{ \begin{array}{r} \texttt{append}(x_1, x_2, x_2) \text{ :- } x_1 \vee (x_2 \wedge x_3) \\ \texttt{rev}(x_1, x_2) \text{ :- } x_1 \end{array} \right\}$$

To illustrate the gfp consider computing the determinacy condition for `rev` in I_2. The first clause of `rev` possesses body atoms which are deterministic builtins. Thus the rigidity condition of *true* is trivially sufficient for this clause to be deterministic. Consider now the second clause. The two calls in its body, `rev(Xs1,Ys1)` and `append(Ys1,Ys2,Ys)`, are processed separately as follows:

- The determinacy condition in I_1 for `rev(Xs1,Ys1)` is $\texttt{Xs1} \vee \texttt{Ys1}$ and the combined success pattern of the equations that precede it is $(\texttt{Xs} \leftrightarrow \texttt{Xs1}) \wedge \texttt{Ys2}$. Thus `rev(Xs1,Ys1)` is deterministic if $((\texttt{Xs} \leftrightarrow \texttt{Xs1}) \wedge \texttt{Ys2}) \rightarrow (\texttt{Xs1} \vee \texttt{Ys1})$ holds upon entry to the body of the clause.
- The determinacy condition in I_1 for `append(Ys1,Ys2,Ys)` is $\texttt{Ys1} \vee (\texttt{Ys2} \wedge \texttt{Ys})$. The combined success pattern of the equations and the call `rev(Xs1,Ys1)` that precede it is $(\texttt{Xs} \leftrightarrow \texttt{Xs1}) \wedge \texttt{Ys2} \wedge (\texttt{Xs1} \wedge \texttt{Ys1}) = \texttt{Xs} \wedge \texttt{Xs1} \wedge \texttt{Ys1} \wedge \texttt{Ys2}$. The call `append(Ys1,Ys2,Ys)` is deterministic if $(\texttt{Xs} \wedge \texttt{Xs1} \wedge \texttt{Ys1} \wedge \texttt{Ys2}) \rightarrow (\texttt{Ys1} \vee (\texttt{Ys2} \wedge \texttt{Ys}))$ holds when the body is entered.

These conditions for determinacy of `rev(Xs1,Ys1)` and `append(Ys1,Ys2,Ys)` are then conjoined to give, say f, (in this case conjunction is trivial since the condition for `append` is *true*). The condition f is formulated in terms of the rigidity of variables occurring in the body of the `rev` clause. What is actually required is a condition on a `rev` call that is sufficient for determinacy. Thus those variables in f that do not occur in the clause head, namely `Xs1`, `Ys1` and `Ys2`, are eliminated from f to obtain a condition sufficient for the call `rev(Xs,Ys)` to be deterministic. Eliminating `Xs1`, `Ys1` and `Ys2` from f (in the manner prescribed in section 4) gives `Xs`. If `Xs` holds, then f holds. Hence if the second clause is selected and the call possesses a rigid first argument, then determinacy follows.

Finally, determinacy conditions for the two `rev` clauses are conjoined with the mutual exclusion condition to obtain $\mathit{true} \wedge \texttt{Xs} \wedge (\texttt{Xs} \vee \texttt{Ys}) = \texttt{Xs}$. Thus the call `rev(Xs,Ys)` is guaranteed to be deterministic if `Xs` is rigid.

3 Synthesising Mutual Exclusion Conditions

To compute mutual exclusion conditions, it is necessary to characterise successful computations at level of clauses. Specifically, it is necessary to characterise the set of solutions for any call that can be obtained with a derivation that commences with a given clause. Success pattern analysis can be adapted to this task.

3.1 Success Patterns

Example 1. To illustrate a success pattern analysis other than argument-size analysis, consider a depth-k analysis of the Quicksort program listed below.

```
(1) sort(X,Y) :- L = [], qsort(X,Y,L).

(2) qsort(X,S,T) :- X = [], S = T.
(3) qsort(X,S,T) :- X = [Y|X1], M1 = [Y|M],
        part(X1,Y,L,G), qsort(L,S,M1), qsort(G,M,T).

(4) part(X,M,L,G) :- Xs = [], L = [], G = [].
(5) part(X,M,L,G) :-
        X = [Y|X1], L = [Y|L1], Y ≤ M, part(X1,M,L1,G).
(6) part(X,M,L,G) :-
        X = [Y|X1], G = [Y|G1], Y > M, part(X1,M,L,G1).
```

In this context of depth-k analysis, a success pattern is an atom paired with a Herbrand or linear constraint where the terms occurring in the constraint have a depth that does not exceed a finite bound k. The success patterns for the predicates and clauses are given below, to the left and the right, respectively.

$$\begin{array}{r} \texttt{sort}(x_1,x_2) \text{ :- } \mathit{true} \\ \texttt{qsort}(x_1,x_2,x_3) \text{ :- } \mathit{true} \\ \texttt{part}(x_1,x_2,x_3,x_4) \text{ :- } \mathit{true} \end{array} \qquad \begin{array}{ll} 1 & \texttt{sort}(x_1,x_2) \text{ :- } \mathit{true} \\ 2 & \texttt{qsort}(x_1,x_2,x_3) \text{ :- } x_1 = [], x_2 = x_3 \\ 3 & \texttt{qsort}(x_1,x_2,x_3) \text{ :- } x_1 = [\_|\_], x_2 = [\_|\_] \\ 4 & \texttt{part}(x_1,x_2,x_3,x_4) \text{ :- } x_1 = [], x_3 = [], x_4 = [] \\ 5 & \texttt{part}(x_1,x_2,x_3,x_4) \text{ :- } x_1 = [y|\_], x_3 = [y|\_], y \leq x_2 \\ 6 & \texttt{part}(x_1,x_2,x_3,x_4) \text{ :- } x_1 = [y|\_], x_4 = [y|\_], y > x_2 \end{array}$$

3.2 Mutual Exclusion Conditions

The technique previously introduced for synthesising mutual exclusion conditions is formulated in terms of argument-size analysis and rigidity analysis and the relationship between rigidity and size; rigidity constraints are used to specify conditions under which pairs of size abstractions are incompatible. To generalise these ideas to other domains, such as depth-k, it is necessary to generalise the concept of norm and replace it with a mapping ν from the set of terms to a set of abstract terms. The concept of rigidity is still meaningful in this general setting: a term t is rigid wrt ν iff $\nu(\theta(t)) = \nu(t)$ for every $\theta \in \text{Sub}$ where Sub is the set of substitutions. Let $rigid_\nu$ be the rigidity predicate on terms induced by ν, that is, $rigid_\nu(t)$ holds if t is rigid wrt ν. For example, if ν is depth-2 abstraction [20] then $rigid_\nu(t)$ holds iff all the variables in t occur at depth 2 or more.

Mutual exclusion conditions are expressed within a dependency domain that can specify rigidity requirements. The property that all the variables in a term occur at or beneath level k can be tracked within the dependency domain, but it is simpler to trace a property that implies rigidity (rather than the induced rigidity property itself). Hence let $rigid'_\nu$ denote any predicate such that $rigid_\nu(t)$ holds if $rigid'_\nu(t)$ holds. For instance, $rigid'_\nu(t) = ground(t)$. Such a predicate

can then induce abstraction $\alpha_{\text{rigid}'_\nu} : \wp(\text{Sub}) \to \text{Pos}$ and concretisation $\gamma_{\text{rigid}'_\nu} : \text{Pos} \to \wp(\text{Sub})$ maps between Pos and $\wp(\text{Sub})$ as follows:

$$\gamma_{\text{rigid}'_\nu}(f) = \{\theta \in \text{Sub} \mid \forall \kappa \in \text{Sub.assign}(\kappa \circ \theta) \models f\}$$
$$\alpha_{\text{rigid}'_\nu}(\Theta) = \wedge\{f \in \text{Pos} \mid \Theta \subseteq \gamma_{\text{rigid}'_\nu}(f)\}$$

where $\text{assign}(\theta) = \wedge\{x \leftrightarrow \text{rigid}'_\nu(\theta(x)) \mid x \in \text{dom}(\theta)\}$. Note that although the underlying domain is Pos – the set of positive Boolean functions – the abstraction is not necessarily groundness. Indeed, the predicate rigid'_ν parameterises the meaning of $\alpha_{\text{rigid}'_\nu}$ and $\gamma_{\text{rigid}'_\nu}$ and these maps define the interpretation of Pos.

Now that a target domain exists in which determinacy conditions can be expressed, it remains to define a general procedure for inferring these conditions. Let $p(\boldsymbol{x})$:-c_1 and $p(\boldsymbol{x})$:-c_2 be the success patterns of two clauses C_1 and C_2 of $p(\boldsymbol{x})$ where c_1 and c_1 are abstract term constraints such as depth-k abstractions. Let $Y \subseteq var(\boldsymbol{x})$. The following predicate checks if the condition $\wedge_{y \in Y} \text{rigid}'_\nu(y)$ is enough for C_1 and C_2 to be mutually exclusive on $p(\boldsymbol{x})$.

$$\mathcal{X}_P(Y, p(\boldsymbol{x}), C_1, C_2) = (\overline{\exists} Y(c_1) \wedge \overline{\exists} Y(c_2) = false)$$

The following proposition (whose proof is given in [15]) formalises the intuitive argument that was given in section 2.2.

Proposition 1. Let $\theta \in \text{Sub}$ and $Y \subseteq var(\boldsymbol{x})$. Suppose $\mathcal{X}_P(Y, p(\boldsymbol{x}), C_1, C_2)$ holds and $\wedge_{y \in Y} rigid'_\nu(\theta(y))$ holds. Then

- all derivations of $\theta(p(\boldsymbol{x}))$ using C_1 as the first clause fail or
- all derivations of $\theta(p(\boldsymbol{x}))$ using C_2 as the first clause fail.

Now let S denote the set of clauses that define $p(\boldsymbol{x})$. A rigidity constraint $\wedge Y$ is a mutual exclusion condition for $p(\boldsymbol{x})$ iff it is a mutual exclusion condition for all pairs of clauses drawn from S. This observation leads to the following:

$$\mathcal{X}_P(p(\boldsymbol{x})) = \bigvee\{\wedge Y \mid Y \subseteq var(\boldsymbol{x}) \wedge \forall C_1, C_2 \in S.(C_1 \neq C_2 \to \mathcal{X}_P(Y, p(\boldsymbol{x}), C_1, C_2)\}$$

The following corollary of proposition 1 verifies that $\mathcal{X}_P(p(\boldsymbol{x}))$ is truly a mutual exclusion condition for $p(x)$.

Corollary 1. If $\alpha_{\text{rigid}'_\nu}(\{\theta\}) \models \mathcal{X}_P(p(\boldsymbol{x}))$ then at most one clause of $p(\boldsymbol{x})$ can lead to a successful derivation of $\theta(p(\boldsymbol{x}))$.

Example 2. The left-hand column gives the exclusion conditions for the quicksort program, synthesised from the clause-level depth-k abstractions listed in example 1 and using the predicate $rigid'_\nu(t) = ground(t)$. The same predicate was used to generate the exclusion conditions in the right-hand column using argument-size abstractions (not provided).

$$\mathcal{X}_P(\texttt{sort}(x_1, x_2)) = true \qquad \mathcal{X}_P(\texttt{sort}(x_1, x_2)) = true$$
$$\mathcal{X}_P(\texttt{qsort}(x_1, x_2, x_3)) = x_1 \qquad \mathcal{X}_P(\texttt{qsort}(x_1, x_2, x_3)) = x_1 \vee (x_2 \leftrightarrow x_3)$$
$$\mathcal{X}_P(\texttt{part}(x_1, x_2, x_3, x_4)) = x_1 \wedge x_2 \qquad \mathcal{X}_P(\texttt{part}(x_1, x_2, x_3, x_4)) = false$$

Note that weaker requirements for mutual exclusion can be obtained by combining these two sets of conditions. Note too that these conditions can only be combined by operating in a domain defined in terms of a common predicate $ground(t)$ which is stronger than both $rigid_{\|.\|}(t)$ and $rigid_{\mathrm{depth}-k}(t)$.

4 Synthesising Determinacy Conditions

This section revisits the backward analysis that strengthens mutual exclusion conditions to obtain the determinacy conditions. As with the previous section, the presentation focusses on those issues left open in the worked example section.

4.1 Abstracting the Program for General Rigidity

The exercise of specifying $\alpha_{\mathrm{rigid}'_\nu}$ and $\gamma_{\mathrm{rigid}'_\nu}$ is more than an aesthetic predilection. It provides a mechanism for deriving an abstract program that captures rigidity relationships between program variables where the notion of rigidity is specified by rigid'_ν. Consider first how to abstract an equation $t_1 = t_2$ in the context of rigid'_ν. The relationship between an equation and its most general unifiers (mgus) is such that the equation can be safely described by any Boolean function f such that $\alpha_{\mathrm{rigid}'_\nu}(\{\theta\}) \models f$ where θ is any mgu of $t_1 = t_2$. For example, if $\nu'(t) = \|t\|$ then $x_1 \leftrightarrow x_3$ describes $x_1 = [x_2|x_3]$. To see this, let $\kappa \in \mathrm{Sub}$ and observe that $\theta = \{x_1 \mapsto [x_2|x_3]\}$ is a mgu of the equation $x_1 = [x_2|x_3]$. Then

$$\mathrm{rigid}'_\nu(\kappa \circ \theta(x_3)) = \mathrm{rigid}'_\nu(\kappa(x_3)) = \mathrm{rigid}'_\nu([\kappa(x_2)|\kappa(x_3)]) = \mathrm{rigid}'_\nu(\kappa \circ \theta(x_1))$$

Thus $\mathrm{assign}(\kappa \circ \theta) \models (x_1 \leftrightarrow x_3)$ for all $\kappa \in \mathrm{Sub}$, whence it follows that $x_1 \leftrightarrow x_3$ describes $x_1 = [x_2|x_3]$. If $\mathrm{rigid}'_\nu(t) = ground(t)$ then $t_1 = t_2$ is abstracted by $\wedge\{x \leftrightarrow \wedge vars(\theta(x)) \mid x \in dom(\theta)\}$ where θ is a mgu of the equation $t_1 = t_2$, though $(\wedge vars(t_1)) \leftrightarrow (\wedge vars(t_2))$ is a simpler, albeit less precise, abstraction. A call to a builtin $p(\boldsymbol{x})$ can be handled by abstracting it with any function f such that $\alpha_{\mathrm{rigid}'_\nu}(\Theta) \models f$ where Θ is the set of computed answers for $p(\boldsymbol{x})$. For instance, if $\nu'(t) = ground(t)$ then $x_1 \wedge x_2$ describes the call $(x_1$ `is` $x_2)$ whereas if $\nu'(t) = \|t\|$ then $x_1 \wedge x_2$ describes the builtin $\texttt{length}(x_1, x_2)$.

Example 3. The following rigidity program is obtained from the quicksort program using $\mathrm{rigid}'_\nu(t) = ground(t)$.

```
(1) sort(X, Y) :- L, qsort(X, Y, L).
(2) qsort(X, S, T) :- X, S ↔ T.
(3) qsort(X, S, T) :- X ↔ (Y ∧ X1), M1 ↔ (Y ∧ M),
        part(X1, Y, L, G), qsort(L, S, M1), qsort(G, M, T).

(4) part(X, M, L, G) :- Xs, L, G.
(5) part(X, M, L, G) :-
        X ↔ (Y ∧ X1), L ↔ (Y ∧ L1), Y, M, part(X1,M,L1,G).
(6) part(X, M, L, G) :-
        X ↔ (Y ∧ X1), G ↔ (Y ∧ G1), Y, M, part(X1,M,L,G1).
```

Example 4. Once the abstract program is defined, the rigidity success patterns can be calculated in the manner previously described to give:

$$\begin{aligned} \texttt{part}(x_1, x_2, x_3, x_4) &\text{ :- } x_1 \wedge x_3 \wedge x_4 \\ \texttt{qsort}(x_1, x_2, x_3) &\text{ :- } x_1 \wedge (x_2 \leftrightarrow x_3) \\ \texttt{sort}(x_1, x_2) &\text{ :- } x_1 \leftrightarrow x_2 \end{aligned}$$

4.2 Determinacy Conditions

Synthesis of determinacy conditions commences by assuming that all calls are trivially determinate, that is, the condition *true* is sufficient for determinacy. These conditions are then checked by reasoning backwards across all clauses. If a condition turns out to be too weak then it is strengthened and the whole process is repeated until the conditions are verified to be sufficient for determinacy. One of the more subtle aspects of this procedure relates to variable elimination. If a condition f, defined in terms of a variable x is sufficient for determinacy, then it can become necessary to calculate another condition, g say, independent of x which is also sufficient for determinacy. Universal quantification operator $\forall_x : \text{Pos} \mapsto \text{Pos}$ provides a mechanism for doing this:

$$\forall_x(f) = \text{if } f' \in \text{Pos then } f' \text{ else } \mathit{false} \quad \text{where} \quad f' = f[x \mapsto \mathit{true}] \wedge f[x \mapsto \mathit{false}]$$

The significance of this operator is that $\forall_x(f) \models f$, hence if f is sufficient for determinacy, then so is $\forall_x(f)$. To succinctly define the gfp operator it is convenient to define a project onto (rather than project out) operator $\overline{\forall}_Y(f)\forall_{y_1}(\forall_{y_2}(\cdots\forall_{y_n}(f)\cdots))$ where each y_i is a (free) variable occurring in f which does not appear in the set of variables Y; in effect f is projected onto Y.

Example 5. Consider $\overline{\forall}_{\{\texttt{X,S,T}\}}(e)$ with $e = \texttt{X} \vee (\texttt{X1} \wedge \texttt{M1}) \vee (\texttt{X1} \wedge \texttt{Y})$. Now

$$e[\texttt{M1} \mapsto \mathit{true}] \wedge e[\texttt{M1} \mapsto \mathit{false}] = (\texttt{X} \vee \texttt{X1}) \wedge (\texttt{X} \vee (\texttt{X1} \wedge \texttt{Y})) = \texttt{X} \vee (\texttt{X1} \wedge \texttt{Y})$$

Thus put $e' = \forall_{\texttt{M1}}(e) = \texttt{X} \vee (\texttt{X1} \wedge \texttt{Y})$ and repeating this tactic:

$$e'[\texttt{X1} \mapsto \mathit{true}] \wedge e'[\texttt{X1} \mapsto \mathit{false}] = (\texttt{X} \vee \texttt{Y}) \wedge (\texttt{X})\texttt{X}$$

Hence put $e'' = \forall_{\texttt{X1}}(e') = \texttt{X}$ and it follows $\overline{\forall}_{\{\texttt{X,S,T}\}}(e)\texttt{X}$. Observe that $\texttt{X} \models e$.

The gfp operates on an abstract program P obtained via rigidity abstraction. To express the operator intuitively, the success set of rigidity patterns, denoted S, is considered to be a map from atoms to Pos formulae. Similarly, the rigidity conditions inferred in the gfp, denoted I, are represented as a map from atoms to formulae. The mechanism that the gfp operator uses to successively update I is to replace each pattern $p(\boldsymbol{x})$:- $f \in I$ with another $p(\boldsymbol{x})$:- $\mathcal{X}_P(p(\boldsymbol{x})) \wedge (\wedge F)$ until stability is achieved where the set of Boolean formula F is defined by:

$$F = \left\{ \overline{\forall}_{\boldsymbol{x}}(e) \;\middle|\; \begin{array}{c} p(\boldsymbol{x}) \text{ :- } f_1, \cdots, f_m, p_1(\boldsymbol{x}_1), \cdots, p_n(\boldsymbol{x}_n) \in P \\ g_i = (\wedge_{1 \le k \le m} f_k) \wedge (\wedge_{1 \le j < i} S(p(\boldsymbol{x}_j))) \\ e = \bigwedge_{1 \le i \le n} (g_i \rightarrow I(p_i(\boldsymbol{x}_i))) \end{array} \right\}$$

The function $\mathcal{X}_P(p(\boldsymbol{x})) \wedge (\wedge F)$ is at least as strong as the formula f it replaces and thus the operator generates a downward iteration sequence. If I' is the gfp thus obtained, the following theorem states how it characterises determinacy.

Theorem 1. *If* $\theta \in \mathrm{Sub}$ *and* $\alpha_{\mathrm{rigid}'_\nu}(\{\theta\}) \models I'(p(\boldsymbol{x}))$ *then* $\theta(p(\boldsymbol{x}))$ *has at most one computed answer.*

Example 6. The iterates that arise when processing the quicksort program are

$$I_0 = \left\{ \begin{array}{rl} \mathtt{part}(x_1,x_2,x_3,x_4) & \text{:- } \mathit{true}, \\ \mathtt{qsort}(x_1,x_2,x_3) & \text{:- } \mathit{true}, \\ \mathtt{sort}(x_1,x_2) & \text{:- } \mathit{true} \end{array} \right\} \quad I_1 = \left\{ \begin{array}{rl} \mathtt{part}(x_1,x_2,x_3,x_4) & \text{:- } x_1 \wedge x_2, \\ \mathtt{qsort}(x_1,x_2,x_3) & \text{:- } x_1, \\ \mathtt{sort}(x_1,x_2) & \text{:- } \mathit{true} \end{array} \right\}$$

$$I_3 = I_2 = \left\{ \begin{array}{rl} \mathtt{part}(x_1,x_2,x_3,x_4) & \text{:- } x_1 \wedge x_2, \\ \mathtt{qsort}(x_1,x_2,x_3) & \text{:- } x_1, \\ \mathtt{sort}(x_1,x_2) & \text{:- } x_1 \end{array} \right\}$$

To illustrate how computation proceeds, consider computing the determinacy condition for `qsort` in I_2. The first abstract clause for $\mathtt{qsort}(\mathtt{X},\mathtt{S},\mathtt{T})$ does not contain any call; hence its determinacy condition is computed as *true*. The second abstract clause for $\mathtt{qsort}(\mathtt{X},\mathtt{S},\mathtt{T})$ has three calls. The first call $\mathtt{part}(\mathtt{X1},\mathtt{Y},\mathtt{L},\mathtt{G})$ has a determinacy condition $\mathtt{X1} \wedge \mathtt{Y}$ in I_1. The cumulative success patterns of the builtins that precede it are $(\mathtt{X} \leftrightarrow (\mathtt{Y} \wedge \mathtt{X1})) \wedge (\mathtt{M1} \leftrightarrow (\mathtt{Y} \wedge \mathtt{M}))$. Thus if

$$e_1 = ((\mathtt{X} \leftrightarrow \mathtt{Y} \wedge \mathtt{X1}) \wedge (\mathtt{M1} \leftrightarrow \mathtt{Y} \wedge \mathtt{M})) \rightarrow (\mathtt{X1} \wedge \mathtt{Y})$$

holds when the body is entered, then $\mathtt{part}(\mathtt{X1},\mathtt{Y},\mathtt{L},\mathtt{G})$ will be deterministic.

The second call $\mathtt{qsort}(\mathtt{L},\mathtt{S},\mathtt{M1})$ has a determinacy condition `L` in I_1 and the success pattern of $\mathtt{part}(\mathtt{X1},\mathtt{Y},\mathtt{L},\mathtt{G})$ is $\mathtt{X1} \wedge \mathtt{L} \wedge \mathtt{G}$. Moreover the Boolean function $f = (\mathtt{X} \leftrightarrow \mathtt{Y}) \wedge (\mathtt{M1} \leftrightarrow (\mathtt{Y} \wedge \mathtt{M})) \wedge \mathtt{X1} \wedge \mathtt{L} \wedge \mathtt{G}$ describes effect of the calls that precede $\mathtt{qsort}(\mathtt{L},\mathtt{S},\mathtt{M1})$. Hence $e_2 = f \rightarrow \mathtt{L}\mathit{true}$ is a condition which, if it holds at entry to the body, is sufficient for the call $\mathtt{qsort}(\mathtt{L},\mathtt{S},\mathtt{M1})$ to be deterministic. Likewise $e_3 = \mathit{true}$ is sufficient for the $\mathtt{qsort}(\mathtt{G},\mathtt{M},\mathtt{T})$ call to be deterministic. The combined determinacy condition is thus $e_1 \wedge e_2 \wedge e_3 = e_1$ and eliminating the body variables which do not occur in the head (using the result from example 5) yields $\overline{\forall}_{\{\mathtt{X},\mathtt{S},\mathtt{T}\}}(e_1)\mathtt{X}$. Combining this with the mutual exclusion condition gives `X`, thus the determinacy requirement for `qsort` does not change.

An astute reader will have noticed in the worked example section that a call to the $\mathtt{rev}(x_1,x_2)$ predicate is determinate if either x_1 *or* x_2 are rigid. Yet the analysis only infers that the rigidity of x_1 is sufficient for determinacy. If the `rev` and `append` calls in the body of the second `rev` clause are interchanged, however, then the analysis will infer that the rigidity of x_2 is sufficient for determinacy. This would suggest the following revision of the above operator: infer a determinacy condition for each permutation of the body atoms; then apply disjunction to merge these conditions to find a more general condition sufficient for determinate behaviour of that clause. However, this tactic, as well as being

benchmark	*predicate*	*exclusion condition*	*determinacy condition*
treesort	`tree_to_list_aux`(x_1, x_2, x_3)	x_1	x_1
	`tree_to_list`(x_1, x_2)	*true*	x_1
	`list_to_tree`(x_1, x_2)	*true*	x_1
	`insert_list`(x_1, x_2, x_3)	x_1	$x_1 \wedge x_2$
	`insert`(x_1, x_2, x_3)	$x_1 \wedge (x_2 \vee x_3)$	$x_1 \wedge (x_2 \vee x_3)$
	`treesort`(x_1, x_2)	*true*	x_1
queens	`noattack`(x_1, x_2, x_3)	x_2	x_2
	`safe`(x_1)	x_1	x_1
	`delete`(x_1, x_2, x_3)	*false*	*false*
	`perm`(x_1, x_2)	$x_1 \vee x_2$	*false*
	`queens`(x_1, x_2)	*true*	*false*
permsort	`select`(x_1, x_2, x_3)	*false*	*false*
	`ordered`(x_1)	x_1	x_1
	`perm`(x_1, x_2)	$x_1 \vee x_2$	*false*
	`sort`(x_1, x_2)	*true*	*false*
serialize	`arrange0`(x_1, x_2)	$x_1 \vee x_2$	x_1
	`numbered`(x_1, x_2, x_3)	x_1	x_1
	`palin`(x_1)	*true*	*true*
	`pairlists`(x_1, x_2, x_3)	$x_1 \vee x_2 \vee x_3$	$x_1 \vee x_2 \vee x_3$
	`serialize0`(x_1, x_2)	*true*	$x_1 \wedge x_2$
	`split0`(x_1, x_2, x_3, x_4)	$x_1 \wedge x_2$	$x_1 \wedge x_2$
	`go`(x_1)	*true*	*false*

Fig. 1. Precision results for determinacy inference

potentially inefficient, is also in general wrong. To see this, suppose that the sufficient condition for determinacy for one goal ordering is x_1 and the condition for another is $x_1 \rightarrow x_2$. However, within Pos, $x_1 \vee (x_1 \rightarrow x_2) = \mathit{true}$ and yet *true* is the vacuous condition which places no constraint on the call `rev`(x_1, x_2).

5 Experimental Evaluation

To evaluate inference analysis, a prototype analyzer has been constructed in SICStus Prolog 3.8.3. The implementation follows sections 3 and 4 closely. The depth-k and argument-size analyses (which applies term-size abstraction) compute success patterns for each clause in the input program, syntheses groundness abstractions sufficient for mutual exclusion. These modules also produce the abstract program on which subsequent analyses are based. The backward analysis is engineered using much of the machinery described in [12]. One notable difference is that some builtins require special handling; most builtins are determinate but others are determinate only when certain conditions are satisfied. For instance, `current_op`(x_1,x_2,x_3) is determinate only when x_2 are x_3 are ground.

The analyser has been applied to 50 programs ranging in size between 10 and 4000 LOC. These programs can be found at `http://www.oakland.edu/~l2lu/Benchmarks-Det.zip`. Quantitative precision measures are difficult for determinacy inference because the number of predicates that the programmer intended to be determinate is, in general, unknown. To demonstrate the precision

	argument-size			*depth-k*				*argument-size*			*depth-k*		
file	*succ*	*lfp*	*gfp*	*succ*	*lfp*	*gfp*	*file*	*succ*	*lfp*	*gfp*	*succ*	*lfp*	*gfp*
boyer	1666	591	60	441	360	50	peep	404	170	30	761	441	70
bryant	7522	261	90	371	321	80	peval	6938	621	100	4466	611	90
chat_80	67393	2153	431	494578	4977	631	press	584	251	40	320	381	70
ga	2146	161	40	2814	290	40	reducer	7867	270	50	190	301	50
ili	2314	450	111	1222	531	100	rubik	30150	420	70	571	3755	221
ime	653	140	40	120	161	40	sim	10270	561	171	35411	611	100
nand	17921	1682	421	841	801	260	sim_v5-2	2948	581	170	491	701	180
nbody	877	191	30	241	301	80	trs	13174	280	91	321	450	100

Fig. 2. Timing results for determinacy inference

of the analysis, some illustrative results are therefore given for several familiar programs. The results for the first program listed in Figure 1 illustrate that the determinacy conditions are often disjunctive reflecting the multi-mode nature of predicates. The second program demonstrates that the analysis will infer *false* for a predicate that is genuinely non-determinate. The third program shows that even the exclusion conditions are themselves interesting. For example, one might have thought that the predicate `select`(x_1, x_2, x_3) – which selects an element x_1 of the list x_2 to give the residual list x_3 – is determinate if called with x_1, x_2 and x_3 ground. However, the call `select(1, [1,1,2], [1,2])` succeeds twice and as a consequence `sort([1,1,2], L)` manifests the buggy behaviour that it generates the answer `L = [1,1,2]` twice. Finally, the fourth program illustrates a so-called false positive. The top-level predicate `go`(x_1) appears to be determinate and we conjecture that this can be inferred by replacing the groundness analysis used in the above experiments with a rigidity analysis [9] that is sensitive to the particular structure of the trees that arise in serialize.

To assess scalability, timing experiments were performed on the analysis components using a 2.49GHz PC with 240 MB RAM running XP. Only the timings for the larger programs are given in Figure 2. The *succ*, *lfp* and *gfp* columns give the time in milliseconds required for the argument-size or depth-k analysis and calculating the lfp and gfp using the exclusion conditions synthesised from one of these analysis (not both together). Interestingly, the gfp is uniformly faster than the lfp despite the fact that the gfp operator is more complicated than the lfp operator. The table shows that the analysis time is dominated by the analyses that feed the client analysis – the backward analysis. There is no reason why the argument size analysis cannot be improved by replacing a constraint based implementation [2] with one based on a polyhedral library [16]. Moreover, the depth-k analysis would benefit from a more intelligent iteration strategy. Nevertheless, the results demonstrate that determinacy inference is practical even when component analyses are implemented naively.

6 Related Work

Giacobazzi and Ricci [10] recognize the value of goal-independence in determinacy analysis and present a solution that tracks deterministic ground depen-

dencies. A ground dependence from a set of input arguments to a set of output arguments is deterministic if, whenever the input arguments are ground, the execution of the predicate binds any output argument to a single ground term. Hence their proposal cannot reason about predicates that compute the same output multiply. The work predates the domains Def and Pos [1] and thus is formulated in terms of hypergraphs. However, even defining an order on hypergraphs is surprisingly subtle. For example, the ordering on abstract atoms proposed in [10] asserts that $p(g, g)$ is less than the atom $p(ng, ng)$ paired with a deterministic ground dependence from the first argument to the second. Observe, however, that the first abstract atom describes a set of concrete atoms that includes $p(a, b)$ and $p(a, c)$ but the set of concrete atoms described by the second cannot include both. Nevertheless, the proposal is not fundamentally flawed and in our opinion the work is in many ways ahead of its time.

Some determinacy checking analyses [14, 17, 19] are rich enough to reason about cuts, if-then-else, and even check for mutual exclusion by applying integer programming. These works raise a number of intriguing questions for determinacy inference, for example, how can cut be accommodated [17, 19] and how hard is determinacy inference [14] (presumably inference is as hard as checking).

Determinacy inference was inspired by the modular construction of termination inference [8] which is itself composed of components that include argument-size analysis [2] and backward analysis [12]. In termination inference, size relations are used to deduce grounding conditions sufficient to observe size decreases, and thus termination, on successive recursive calls. Backward analysis is applied to derive sufficient conditions for termination for a compound goal that is executed left-to-right. In determinacy inference, the size issue does not relate primarily to calls but to the relative sizes of the answers generated from different clauses. Determinacy inference also differs from termination inference in that the latter applies the framework of [12] directly whereas the former does not. While the gfp in [12] propagates requirements from right-to-left across the body of a clause; the gfp in this paper propagates determinacy requirements on each call in the body using the conjunction of the success patterns of those calls that precede it. Observe that this conjunction can be pre-computed; it does not need to be reevaluated on each application of gfp operator. Thus, the structure of the gfp presented in this paper enables efficiency savings.

7 Conclusions

This paper has shown how the problem of checking that a goal is determinate can be reformulated as the problem of inferring a class of determinate goals. Despite the generality of this problem, this paper has shown how a determinate inference engine can be constructed by composing classic goal-independent success set analyses such as argument-size and depth-k analysis with modern backward analysis techniques. The paper has demonstrated that the analysis is tractable and the importance of determinacy suggests that the analysis will be useful.

Acknowledgements. This work was supported, in part, by NSF grants CCR-0131862 and INT-0327760. The authors are grateful to John Gallagher for gently guiding them through the binding-time analysis literature. The authors have also benefited from discussions on backward analysis with Samir Genaim and the insightful comments of the referees.

References

1. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two Classes of Boolean Functions for Dependency Analysis. *Science of Computer Programming*, 31(1):3–45, 1998.
2. F. Benoy and A. King. Inferring Argument Size Relationships with CLP($\mathscr{R}$). In *LOPSTR*, volume 1207 of *LNCS*, pages 204–223. Springer-Verlag, 1997.
3. C. Braem, B. Le Charlier, S. Modar, and P. Van Hentenryck. Cardinality Analysis of Prolog. In M. Bruynooghe, editor, *ISLP*, pages 457–471. MIT Press, 1994.
4. M. Bruynooghe, M. Leuschel, and K. Sagonas. A Polyvariant Binding-Time Analysis for Off-line Partial Deduction. In *ESOP*, volume 1381 of *LNCS*, pages 27–41. Springer-Verlag, 2000.
5. S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Extracting Determinacy in Logic Programs. In *ICLP*, pages 424–438. MIT Press, 1993.
6. S. K. Debray and N.-W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.
7. J. P. Gallagher and L. Lafave. Regular Approximation of Computation Paths in Logic and Functional Languages. In *International Seminar on Partial Evaluation*, volume 1110 of *LNCS*, pages 115–136. Springer-Verlag, 1996.
8. S. Genaim and M. Codish. Inferring Termination Conditions for Logic Programs using Backwards Analysis. *TPLP*, 5(1&2):75–91, 2005.
9. R. Giacobazzi, S. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *JLP*, 25(3):191–248, 1995.
10. R. Giacobazzi and L. Ricci. Detecting Determinate Computation by Bottom-Up Abstract Interpretation. In *ESOP*, volume 582 of *LNCS*, pages 167–181. Springer-Verlag, 1992.
11. F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the Mercury compiler. In *Proceedings of the Australian Computer Science Conference*, pages 337–346, 1996.
12. A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *TPLP*, 2:517–547, 2002.
13. M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline Specialisation in Prolog Using a Hand-Written Compiler Generator. *TPLP*, 4(1):139–191, 2004.
14. P. López-García, F. Bueno, and M. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type information. In *Pre-proceedings of LOPSTR*, pages 19–29, 2004.
15. L. Lu and A. King. Determinacy Inference for Logic Programs. Technical Report 19-04, Computing Laboratory, University of Kent, CT2 7NF, 2004.
16. F. Mesnard and R. Bagnara. cTI: A constraint-based termination inference tool for ISO-Prolog. *TPLP*, 5(1&2):243–257, 2005.
17. T. Mogensen. A Semantics-Based Determinacy Analysis for Prolog with Cut. In *Ershov Memorial Conference*, volume 1181 of *LNCS*, pages 374–385. Springer-Verlag, 1996.

18. R. A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.
19. D. Sahlin. Determinacy Analysis for Full Prolog. In *PEPM*, pages 23–30, 1991. SIGPLAN Notices 26(9).
20. T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.
21. M. H. van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, 1976.

Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis*

Oukseh Lee[1], Hongseok Yang[2], and Kwangkeun Yi[3]

[1] Dept. of Computer Science & Engineering, Hanyang University, Korea
[2] ERC-ACI, Seoul National University, Korea
[3] School of Computer Science & Engineering, Seoul National University, Korea

Abstract. We present a program analysis that can automatically discover the shape of complex pointer data structures. The discovered invariants are, then, used to verify the absence of safety errors in the program, or to check whether the program preserves the data consistency. Our analysis extends the shape analysis of Sagiv et al. with grammar annotations, which can precisely express the shape of complex data structures. We demonstrate the usefulness of our analysis with binomial heap construction and the Schorr-Waite tree traversal. For a binomial heap construction algorithm, our analysis returns a grammar that precisely describes the shape of a binomial heap; for the Schorr-Waite tree traversal, our analysis shows that at the end of the execution, the result is a tree and there are no memory leaks.

1 Introduction

We show that a static program analysis can automatically verify pointer programs, such as binomial heap construction and the Schorr-Waite tree traversal. The verified properties are: for a binomial heap construction algorithm, our analysis verifies that the returned heap structure is a binomial heap; for the Schorr-Waite tree traversal, it verifies that the output tree is a binary tree, and there are no memory leaks. In both cases, the analysis took less than 0.2 second in Intel Pentium 3.0C with 1GB memory, and its result is simple and human-readable.

Note that although these programs handle regular heap structures such as binomial heaps and trees, the topology of pointers (e.g., cycles) and their imperative operations (e.g., pointer swapping) are fairly challenging for fully automatic static verification without any annotation from the programmer.

The static analysis is an extension to Sagiv et al.'s shape analysis [13] by grammars. To improve accuracy, we associate grammars, which finitely summarize run-time heap structures, with the summary nodes of the shape graphs. This

* Lee and Yi were supported by the Brain Korea 21 project in 2004, and Yang was supported by R08-2003-000-10370-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 124–140, 2005.

enrichment of shape graph by grammars provides an ample space for precisely capturing the imperative effects on heap structures. The grammar is unfolded to expose an exact heap structure on demand. The grammar is also folded to replace an exact heap structure by an abstract nonterminal. To ensure the termination of the analysis, the grammar merges multiple production rules into a single one, and unify multiple nonterminals; this simplification makes the grammar size remain within a practical bound.

The analysis's correctness is proved via separation logic [12, 11]. The analysis is a composition of abstract operations over the grammar-based shape graphs. The semantics (concretization) of the shape graphs is defined as assertions in separation logic. Each abstract operator is proved safe by showing that the separation-logic assertion for the input graph implies that for the output graph. The input program C wrapped by the input and output assertions $\{P\}C\{Q\}$ is always a provable Hoare triple by the separation-logic proof rules.

The main limitation of our analysis is that the analysis cannot handle DAGs and general graphs. To overcome this limitation, we need to use a more general grammar, where the nonterminals can talk about shared cells.

Related Work. We borrowed several interesting ideas from the shape analysis [14]. Our analysis represents a program invariant using a set of shape graphs where each shape graph consists of either concrete or abstract nodes. It uses the idea of refining an abstract node, often called focus or materialization, and also the idea of merging the shape graphs which have a similar structure [9, 5].

The difference is the use of grammar; it is the main reason for the improved precision of our analysis. Another difference is that our analysis separates node-summarizing criteria from the properties of the summary nodes. Normally, the shape analysis of Sagiv et al. partitions all the concrete nodes according to the instrumentation predicates that they satisfy, and groups each partition into a single summary node. Thus, two different summary nodes must satisfy different sets of instrumentation predicates. Our analysis, on the other hand, groups the concrete nodes using the most approximate grammar: each group is a maximal set of concrete nodes that can be expressed by the most approximate grammar. Then, the analysis summarizes each group by a single summary node, and annotates the summary node with a new grammar that "best" describes the pointer structure of the summarized concrete nodes. As a consequence, two different summary nodes in our analysis can have the identical grammar annotations.

Graph type [7, 10] and shape type [6] are also closely related to our work. Both of them express invariants of heap objects (or data structures) using grammar-based languages, which are more expressive than the grammars we used. However, they assume that all the loop invariants of a program are provided, while our work infers such invariants.

Outline. Section 2 describes the source programming language. Section 3 overviews separation logic that we use to give the meaning of abstract values. Then, we explain the key ideas of our analysis, using a simpler version that can handle tree-like structures with no shared nodes. Section 4 and 5 explain the

abstract domain and abstract operators, and Section 6 defines the analyzer. The simpler version is extended to the full analysis in Section 7. Section 8 demonstrates the accuracy of our analysis using binomial heap construction algorithm and the Schorr-Waite tree traversing algorithm.

2 Programming Language

We use the standard while language with additional pointer operations.

$$\begin{array}{lll} \textit{Vars} \quad x \qquad \textit{Fields} & f & \in \{0,1\} \\ \textit{Boolean Expressions} & B & ::= x = y \mid \texttt{!}B \\ \textit{Commands} & C & ::= x := \texttt{nil} \mid x := y \mid x := \texttt{new} \mid x := y\texttt{->}f \mid x\texttt{->}f := y \\ & & \quad \mid C\,; C \mid \texttt{if}\ B\ C\ C \mid \texttt{while}\ B\ C \end{array}$$

This language assumes that every heap cell is binary, having fields 0 and 1. A heap cell is allocated by `x := new`, and the contents of such an allocated cell is accessed by the field-dereference operation `->`. All the other constructs in the language are standard.

3 Separation Logic with Recursive Predicates

Let *Loc* and *Val* be unspecified infinite sets such that $\text{nil} \notin \mathit{Loc}$ and $\mathit{Loc} \cup \{\text{nil}\} \subseteq \mathit{Val}$. We consider separation logic for the following semantic domains.

$$\mathit{Stack} \triangleq \mathit{Vars} \rightharpoonup_{\text{fin}} \mathit{Val} \qquad \mathit{Heap} \triangleq \mathit{Loc} \rightharpoonup_{\text{fin}} \mathit{Val} \times \mathit{Val} \qquad \mathit{State} \triangleq \mathit{Stack} \times \mathit{Heap}$$

This domain implies that a state has the stack and heap components, and that the heap component of a state has finitely many binary cells.

The assertion language in separation logic is given by the following grammar:[1]

$$P ::= E = E \mid \texttt{emp} \mid (E \mapsto E, E) \mid P * P \mid \texttt{true} \mid P \wedge P \mid P \vee P \mid \neg P \mid \forall x.\, P$$

Separating conjunction $P{*}Q$ is the most important, and it expresses the splitting of heap storage; $P{*}Q$ means that the heap can be split into two parts, so that P holds for the one and Q for the other. We often use precise equality and iterated separating conjunction, both of which we define as syntactic sugars. Let X be a finite set $\{x_1, \ldots, x_n\}$ where all x_i's are different.

$$E \doteq E' \ \triangleq\ E{=}E' \wedge \texttt{emp} \qquad \bigodot_{x \in X} A_x \ \triangleq\ \text{if } (X = \emptyset) \text{ then } \texttt{emp} \text{ else } (A_{x_1} * \ldots * A_{x_n})$$

In this paper, we use the extension of the basic assertion language with recursive predicates [15]:

$$P ::= \ldots \mid \alpha(E, \ldots, E) \mid \mathsf{rec}\ \Gamma\ \mathsf{in}\ P \qquad \Gamma ::= \alpha(x_1, \ldots, x_n) = P \mid \Gamma, \Gamma$$

[1] The assertion language also has the adjoint $-\!\!*$ of $*$. But this adjoint is not used in this paper, so we omit it here.

The extension allows the definition of new recursive predicates by least-fixed points in "rec Γ in P", and the use of such defined recursive predicates in $\alpha(E, \ldots, E)$. To ensure the existence of the least-fixed point in rec Γ in P, we will consider only well-formed Γ where all recursively defined predicates appear in positive positions.

A recursive predicate in this extended language means a set of heap objects. A *heap object* is a pair of location (or locations) and heap. Intuitively, the first component denotes the starting address of a data structure, and the second the cells in the data structure. For instance, when a linked list is seen as a heap object, the location of the head of the list becomes the first component, and the cells in the linked list the second component.

The precise semantics of this assertion language is given by a forcing relation $\models$. For a state (s, h) and an environment η for recursively defined predicates, we define inductively when an assertion P holds for (s, h) and η. We show the sample clauses below; the full definition appears in [8].

$$\begin{array}{ll} (s,h),\eta \models \alpha(E) & \text{iff } \ (\llbracket E \rrbracket s, h) \in \eta(\alpha) \\ (s,h),\eta \models \mathsf{rec}\ \alpha(x){=}P\ \mathsf{in}\ Q & \text{iff } \ (s,h),\eta[\alpha{\rightarrow}k] \models P \\ (\text{where } k = \mathrm{lfix}\, \lambda k_0.\{(v',h') \mid (s[x{\rightarrow}v'],h'),\eta[\alpha{\rightarrow}k_0] \models P\}) . & \end{array}$$

4 Abstract Domain

Shape Graph. Our analysis interprets a program as a (nondeterministic) transformer of *shape graphs*. A shape graph is an abstraction of concrete states; this abstraction maintains the basic "structure" of the state, but abstracts away all the other details. For instance, consider a state $([x{\rightarrow}1, y{\rightarrow}3], [1{\rightarrow}\langle 2, \mathrm{nil}\rangle, 2{\rightarrow}\langle \mathrm{nil}, \mathrm{nil}\rangle, 3{\rightarrow}\langle 1, 3\rangle])$. We obtain a shape graph from this state in two steps. First, we replace the specific addresses, such as 1 and 2, by *symbolic locations*; we introduce symbols a, b, c, and represent the state by $([x{\rightarrow}a, y{\rightarrow}c], [a{\rightarrow}\langle b, \mathrm{nil}\rangle, b{\rightarrow}\langle \mathrm{nil}, \mathrm{nil}\rangle, c{\rightarrow}\langle a, c\rangle])$. Note that this process abstracts away the specific addresses and just keeps the relationship between the addresses. Second, we abstract heap cells a and b by a grammar. Thus, this step transforms the state to $([x{\rightarrow}a, y{\rightarrow}c], [a{\rightarrow}\mathsf{tree}, c{\rightarrow}\langle a, c\rangle])$ where $a{\rightarrow}\mathsf{tree}$ means that a is the address of the root of a tree, whose structure is summarized by grammar rules for nonterminal tree.

The formal definition of a shape graph is given as follows:

$$\begin{array}{l} \mathit{SymLoc} \triangleq \{a, b, c, \ldots\} \qquad \mathit{NonTerm} \triangleq \{\alpha, \beta, \gamma, \ldots\} \\ \mathit{Graph} \triangleq (\mathit{Vars} \rightharpoonup_{\mathrm{fin}} \mathit{SymLoc}) \times (\mathit{SymLoc} \rightharpoonup_{\mathrm{fin}} \{\mathrm{nil}\} + \mathit{SymLoc}^2 + \mathit{NonTerm}) \end{array}$$

Here the set of nonterminals is disjoint from *Vars* and *SymLoc*; these nonterminals represent recursive heap structures such as tree or list. Each shape graph has two components (s, g). The first component s maps stack variables to symbolic locations. The other component g describes heap cells reachable from each symbolic location. For each a, either no heap cells can be reached from a, i.e, $g(a) = \mathrm{nil}$; or, a is a binary cell with contents $\langle b, c\rangle$; or, the cells reachable from a form a heap object specified by a nonterminal α. We also require that g describes

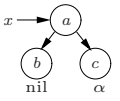
all the cells in the heap; for instance, if g is the empty partial function, it means the empty heap.

The semantics (or concretization) of a shape graph (s, g) is given by a translation into an assertion in separation logic:

$$\mathsf{means}_{\mathrm{v}}(a, \mathrm{nil}) \triangleq a \doteq \mathrm{nil} \quad \mathsf{means}_{\mathrm{v}}(a, \alpha) \triangleq \alpha(a) \quad \mathsf{means}_{\mathrm{v}}(a, \langle b, c\rangle) \triangleq (a \mapsto b, c)$$
$$\mathsf{means}_{\mathrm{s}}(s, g) \triangleq \exists \boldsymbol{a}.\ (\bigodot_{x \in \mathrm{dom}(s)} x \doteq s(x)) * (\bigodot_{a \in \mathrm{dom}(g)} \mathsf{means}_{\mathrm{v}}(a, g(a)))$$

The translation function $\mathsf{means}_{\mathrm{s}}$ calls a subroutine $\mathsf{means}_{\mathrm{v}}$ to get the translation of the value of $g(a)$, and then, it existentially quantifies all the symbolic locations appearing in the translation. For instance, $\mathsf{means}_{\mathrm{s}}([x{\to}a, y{\to}c], [a{\to}\mathsf{tree}, c{\to}\langle a, c\rangle])$ is $\exists ac.\, (x \doteq a) * (y \doteq c) * \mathsf{tree}(a) * (c \mapsto a, c)$.

When we present a shape graph, we interchangeably use the set notation and a graph picture. Each variable or symbolic location becomes a node in a graph, and s and g are represented by edges or annotations. For instance, we draw a shape graph $(s, g) = ([x{\to}a], [a{\to}\langle b, c\rangle, b{\to}\mathrm{nil}, c{\to}\alpha])$ as:



Note that pair $g(a)$ is represented by two edges (the left one is for field 0 and the right one for field 1), and non-pair values $g(b)$ and $g(c)$ by annotations to the nodes.

Grammar. A grammar gives the meaning of nonterminals in a shape graph. We define a *grammar* R as a finite partial function from nonterminals (the lhs of production rules) to $\wp_{\mathsf{nf}}(\{\mathrm{nil}\} + (\{\mathrm{nil}\} + \mathit{NonTerm})^2)$ (the rhs of production rules), where $\wp_{\mathsf{nf}}(X)$ is the family of all nonempty finite subsets of X.

$$\mathit{Grammar} \triangleq \mathit{NonTerm} \rightharpoonup_{\mathrm{fin}} \wp_{\mathsf{nf}}(\{\mathrm{nil}\} + (\{\mathrm{nil}\} + \mathit{NonTerm})^2)$$

Set $R(\alpha)$ contains all the possible shapes of heap objects for α. If $\mathrm{nil} \in R(\alpha)$, α can be the empty heap object. If $\langle \beta, \gamma\rangle \in R(\alpha)$, then some heap object for α can be split into a root cell, the left heap object β, and the right heap object γ. For instance, if $R(\mathsf{tree}) = \{\mathrm{nil}, \langle \mathsf{tree}, \mathsf{tree}\rangle\}$ (i.e., in the production rule notation, $\mathsf{tree} ::= \mathrm{nil} \mid \langle \mathsf{tree}, \mathsf{tree}\rangle$), then tree represents binary trees. In our analysis, we use only *well-formed* grammars, where all nonterminals appearing in the range of a grammar are defined in the grammar.

The meaning $\mathsf{means}_{\mathrm{g}}(R)$ of a grammar R is given by a recursive predicate declaration Γ in separation logic. Γ is defined exactly for $\mathrm{dom}(R)$, and satisfies the following: when $\mathrm{nil} \notin R(\alpha)$, $\Gamma(\alpha)$ is

$$\alpha(a) = \bigvee_{\langle v, w\rangle \in R(\alpha)} \exists bc.\, (a{\mapsto}b, c) * \mathsf{means}_{\mathrm{v}}(b, v) * \mathsf{means}_{\mathrm{v}}(c, w),$$

where neither b nor c appears in a, v or w; otherwise, $\Gamma(\alpha)$ is identical as above except that $a \doteq \mathrm{nil}$ is added as a disjunct. For instance, $\mathsf{means}_{\mathrm{g}}([\mathsf{tree}{\to}\{\mathrm{nil}, \langle \mathsf{tree}, \mathsf{tree}\rangle\}])$ is $\{\mathsf{tree}(a) = a{\doteq}\mathrm{nil} \vee \exists bc.(a \mapsto b, c){*}\mathsf{tree}(b){*}\mathsf{tree}(c)\}$.

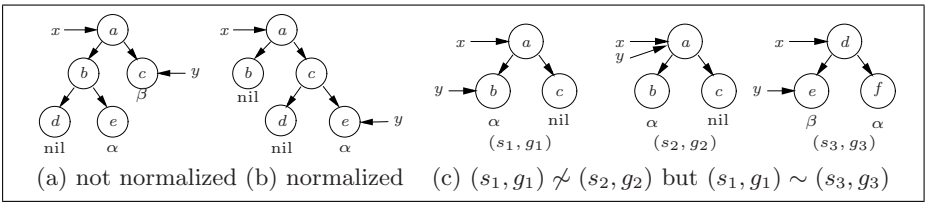


Fig. 1. Examples of the Normalized Shape Graphs and Similar Shape Graphs

Abstract Domain. The abstract domain $\widehat{D}$ for our analysis consists of pairs of a shape graph set and a grammar: $\widehat{D} \stackrel{\Delta}{=} \{\top\} + \wp_{\mathsf{nf}}(\mathit{Graph}) \times \mathit{Grammar}$. The element $\top$ indicates that our analysis fails to produce any meaningful results for a given program because the program has safety errors, or the program uses data structures too complex for our analysis to capture. The meaning of each abstract state $(\mathcal{G}, R)$ in $\widehat{D}$ is $\mathsf{means}(\mathcal{G}, R) \stackrel{\Delta}{=} \mathsf{rec}\ \mathsf{means}_{\mathrm{g}}(R)\ \mathsf{in} \bigvee_{(s,g)\in\mathcal{G}} \mathsf{means}_{\mathrm{s}}(s, g)$.

5 Normalized Abstract States and Normalization

The main strength of our analysis is to automatically discover a grammar that describes, in an "intuitive" level, invariants for heap data structures, and to abstract concrete states according to this discovered grammar. This inference of high-level grammars is mainly done by the normalization function from $\widehat{D}$ to a subdomain $\widehat{D}^{\nabla}$ of *normalized abstract states*. In this section, we explain these two notions, normalized abstract states and normalization function.

5.1 Normalized Abstract States

An abstract state $(\mathcal{G}, R)$ is normalized if it satisfies the following two conditions. First, all the shape graphs (s, g) in $\mathcal{G}$ are abstract enough: all the recognizable heap objects are replaced by nonterminals. Note that this condition on $(\mathcal{G}, R)$ is about individual shape graphs in $\mathcal{G}$. We call a shape graph *normalized* if it satisfies this condition. Second, an abstract state does not have redundancies: all shape graphs are not *similar*, and all nonterminals have *non-similar* definitions.

Normalized Shape Graphs. A shape graph is normalized when it is "maximally" folded. A symbolic location a is *foldable* in (s, g) if $g(a)$ is a pair and there is no path from a to a *shared* symbolic location that is referred more than once. When $\mathrm{dom}(g)$ of a shape graph (s, g) does not have any foldable locations, we say that (s, g) is *normalized*. For instance, Figure 1.(a) is not normalized, because b is foldable: b is a pair and does not reach any shared symbolic locations. On the other hand, Figure 1.(b) is normalized, because all the pairs in the graph (i.e., a and c) can reach shared symbolic location e.

Similarity. We define three notions of similarity: one for shape graphs, another for two cases of the grammar definitions, and the third for the grammar definitions of two nonterminals.

Two shape graphs are similar when they have the similar structures. Let S be a substitution that renames symbolic locations. Two shape graphs (s, g) and (s', g') are *similar up to* S, denoted $(s, g) \sim_S^G (s', g')$, if and only if

1. $\mathrm{dom}(s) = \mathrm{dom}(s')$ and $S(\mathrm{dom}(g)) = \mathrm{dom}(g')$;
2. for all $x \in \mathrm{dom}(s)$, $S(s(x)) = s'(x)$; and
3. for all $a \in \mathrm{dom}(g)$, if $g(a)$ is a pair $\langle b, c\rangle$ for some b and c, then $g'(S(a)) = \langle S(b), S(c)\rangle$; if $g(a)$ is not a pair, neither is $g'(S(a))$.

Intuitively, two shape graphs are S-similar, when equating nil and all nonterminals makes the graphs identical up to renaming S. We say that (s, g) and (s', g') are similar, denoted $(s, g) \sim (s', g')$, if and only if there is a renaming relation S such that $(s, g) \sim_S^G (s', g')$. For instance, in Figure 1.(c), (s_1, g_1) and (s_2, g_2) are not similar because we cannot find a renaming substitution S such that $S(s_1(x)) = S(s_1(y))$ (condition 2). However, (s_1, g_1) and (s_3, g_3) are similar because a renaming substitution $\{d/a, e/b, f/c\}$ makes (s_1, g_1) identical to (s_3, g_3) when nil and all nonterminals are erased from the graphs.

Cases e_1 and e_2 in the grammar definitions are similar, denoted $e_1 \sim^C e_2$, if and only if either both e_1 and e_2 are pairs, or they are both non-pair values. The similarity $E_1 \sim^D E_2$ between grammar definitions E_1 and E_2 uses this case similarity: $E_1 \sim^D E_2$ if and only if, for all cases e in E_1, E_2 has a similar case e' to e ($e \sim^C e'$), and vice versa. For example, in the grammar

$$\alpha ::= \langle \beta, \mathrm{nil}\rangle\,, \quad \beta ::= \mathrm{nil} \mid \langle \beta, \mathrm{nil}\rangle\,, \quad \gamma ::= \langle \gamma, \gamma\rangle \mid \langle \alpha, \mathrm{nil}\rangle$$

the definitions of α and γ are similar because both $\langle \gamma, \gamma\rangle$ and $\langle \alpha, \mathrm{nil}\rangle$ are similar to $\langle \beta, \mathrm{nil}\rangle$. But the definitions of α and β are not similar since α does not have a case similar to nil.

Definition 1 (Normalized Abstract States). *An abstract state* $(\mathcal{G}, R)$ *is* normalized *if and only if*

1. *all shape graphs in* $\mathcal{G}$ *are normalized;*
2. *for all* (s_1, g_1), $(s_2, g_2) \in \mathcal{G}$, *we have* $(s_1, g_1) \sim (s_2, g_2) \Rightarrow (s_1, g_1) = (s_2, g_2)$;
3. *for all* $\alpha \in \mathrm{dom}(R)$ *and all cases* $e_1, e_2 \in R(\alpha)$, $e_1 \sim^C e_2$ *implies that* $e_1 = e_2$;
4. *for all* α, β *in* $\mathrm{dom}(R)$, $R(\alpha) \sim^D R(\beta)$ *implies that* $\alpha = \beta$.

We write $\widehat{D}^\nabla$ for the set of normalized abstract states.

***k*-Bounded Normalized States.** Unfortunately, the normalized abstract domain $\widehat{D}^\nabla$ does not ensure the termination of the analysis, because it has infinite chains. For each number k, we say that an abstract state $(\mathcal{G}, R)$ is k-bounded iff all the shape graphs in $\mathcal{G}$ use at most k symbolic locations, and we define $\widehat{D}_k^\nabla$ to be the set of k-bounded normalized abstract states. This finite domain $\widehat{D}_k^\nabla$ is used in our analysis.

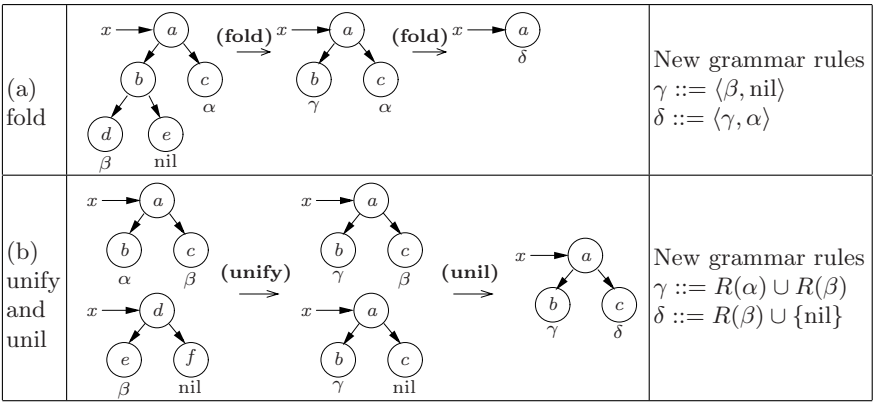


Fig. 2. Examples of (**fold**), (**unify**), and (**unil**)

5.2 Normalization Function

The normalize function transforms $(\mathcal{G}, R)$ to a normalized $(\mathcal{G}', R')$ with a further abstraction (i.e., $\mathsf{means}(\mathcal{G}, R) \Rightarrow \mathsf{means}(\mathcal{G}', R')$).[2] It is defined by the composition of five subroutines: $\mathsf{normalize} = \mathsf{bound}^k \circ \mathsf{simplify} \circ \mathsf{unify} \circ \mathsf{fold} \circ \mathsf{rmjunk}$.

The first subroutine rmjunk removes all the "imaginary" sharing and garbage due to constant symbolic locations, so that it makes the real sharing and garbage easily detectable in syntax. The subroutine rmjunk applies the following two rules until an abstract state does not change. In the definition, "$\uplus$" is a disjoint union of sets, and "$\cdot$" is a union of partial maps with disjoint domains.

(**alias**) $(\mathcal{G} \uplus \{(s \cdot [x{\to}a], g \cdot [a{\to}\mathrm{nil}])\}, R) \rightsquigarrow (\mathcal{G} \cup \{(s \cdot [x{\to}a'], g \cdot [a{\to}\mathrm{nil}, a'{\to}\mathrm{nil}])\}, R)$
where a should appear in (s, g) and a' is fresh.

(**gc**) $(\mathcal{G} \uplus \{(s, g \cdot [a{\to}\mathrm{nil}])\}, R) \rightsquigarrow (\mathcal{G} \cup \{(s, g)\}, R)$ where a does not appear in (s, g)

For instance, given a shape graph $([x{\to}a, y{\to}a], [a{\to}\mathrm{nil}, c{\to}\mathrm{nil}])$, (**gc**) collects the "garbage" c, and (**alias**) eliminates the "imaginary sharing" between x and y by renaming a in $y{\to}a$. So, the shape graph becomes $([x{\to}a, y{\to}b], [a{\to}\mathrm{nil}, b{\to}\mathrm{nil}])$.

The second subroutine fold converts a shape graph to a normal form, by replacing all foldable symbolic locations by nonterminals. The subroutine fold repeatedly applies the following rule until the abstract state does not change:

(**fold**) $(\mathcal{G} \uplus \{(s, g{\cdot}[a{\to}\langle b, c\rangle, b{\to}v, c{\to}v'])\}, R) \rightsquigarrow (\mathcal{G} \cup \{(s, g{\cdot}[a{\to}\alpha])\}, R{\cdot}[\alpha{\to}\{\langle v, v'\rangle\}])$
where neither b nor c appears in (s, g), α is fresh, and v and v' are not pairs.

The rule recognizes that the symbolic locations b and c are accessed only via a. Then, it represents cell a, plus the reachable cells from b and c by a nonterminal α. Figure 2 shows how the (**fold**) folds a tree.

The third subroutine unify merges two similar shape graphs in $\mathcal{G}$. Let (s, g) and (s', g') be similar shape graphs by the identity renaming Δ (i.e., $(s, g) \sim^{G}_{\Delta}$

[2] The normalize function is a reminiscent of the widening in [2, 3].

$(s', g'))$. Then, these two shape graphs are almost identical; the only exception is when $g(a)$ and $g'(a)$ are nonterminals or nil. unify eliminates all such differences in two shape graphs; if $g(a)$ and $g'(a)$ are nonterminals, then unify changes g and g', so that they map a to the same fresh nonterminal γ, and then it defines γ to cover both α and β. The unify procedure applies the following rules to an abstract state $(\mathcal{G}, R)$ until the abstract state does not change:

(**unify**) $(\mathcal{G} \uplus \{(s_1, g_1 \cdot [a_1 \to \alpha_1]), (s_2, g_2 \cdot [a_2 \to \alpha_2])\},\ R)$
$\rightsquigarrow (\mathcal{G} \cup \{(S(s_1), S(g_1) \cdot [a_2 \to \beta]), (s_2, g_2 \cdot [a_2 \to \beta])\},\ R \cdot [\beta \to R(\alpha_1) \cup R(\alpha_2)])$
where $(s_1, g_1 \cdot [a_1 \to \alpha_1]) \sim_S^G (s_2, g_2 \cdot [a_2 \to \alpha_2])$, $S(a_1) \equiv a_2$, $\alpha_1 \not\equiv \alpha_2$, and β is fresh.

(**unil**) $(\mathcal{G} \uplus \{(s_1, g_1 \cdot [a_1 \to \alpha]), (s_2, g_2 \cdot [a_2 \to \text{nil}])\},\ R)$
$\rightsquigarrow (\mathcal{G} \cup \{(S(s_1), S(g_1) \cdot [a_2 \to \beta]), (s_2, g_2 \cdot [a_2 \to \beta])\},\ R \cdot [\beta \to R(\alpha) \cup \{\text{nil}\}])$
where $(s_1, g_1 \cdot [a_1 \to \alpha]) \sim_S^G (s_2, g_2 \cdot [a_2 \to \text{nil}])$, $S(a_1) \equiv a_2$, and β is fresh.

The (**unify**) rule recognizes two similar shape graphs that have different nonterminals at the same position, and replaces those nonterminals by fresh nonterminal β that covers the two nonterminals. The (**unil**) rule deals with the two similar graphs that have, respectively, nonterminal and nil at the same position. For instance, in Figure 2.(b), the left two shape graphs are unified by (**unify**) and (**unil**). We first replace the left children α and β by γ that covers both; that is, to a given grammar R, we add $[\gamma \to R(\alpha) \cup R(\beta)]$. Then we replace the right children β and nil by δ that covers both.

The fourth subroutine simplify reduces the complexity of grammar by combining similar cases or similar definitions.[3] It applies three rules repeatedly:

- If the definition of a nonterminal has two similar cases $\langle \beta, v \rangle$ and $\langle \beta', v' \rangle$, and β and β' are different nonterminals, unify nonterminals β and β'. Apply the same for the second field.
- If the definition of a nonterminal has two similar cases $\langle \beta, v \rangle$ and $\langle \text{nil}, v' \rangle$, add the nil case to $R(\beta)$ and replace $\langle \text{nil}, v' \rangle$ by $\langle \beta, v' \rangle$. Apply the same for the second field.
- If the definitions of two nonterminals are similar, unify the nonterminals.

Formally, the three rules are:

(**case**) $(\mathcal{G}, R) \rightsquigarrow (\mathcal{G}, R)\{\beta/\alpha\}$ where $\{\langle \alpha, v \rangle, \langle \beta, v' \rangle\} \subseteq R(\gamma)$ and $\alpha \not\equiv \beta$.
(same for the second field)

(**nil**) $(\mathcal{G}, R \cdot [\alpha \to E \uplus \{\langle \beta, v \rangle, \langle \text{nil}, v' \rangle\}]) \rightsquigarrow (\mathcal{G}, R'[\beta \to R'(\beta) \cup \{\text{nil}\}])$
where $R' = R \cdot [\alpha \to E \cup \{\langle \beta, v \rangle, \langle \beta, v' \rangle\}]$. (same for the second field)

(**def**) $(\mathcal{G}, R) \rightsquigarrow (\mathcal{G}, R)\{\beta/\alpha\}$ where $R(\alpha) \sim R(\beta)$ and $\alpha \not\equiv \beta$.

Here, $(\mathcal{G}, R)\{\alpha/\beta\}$ substitutes α for β, and in addition, it removes the definition of β from R and re-defines α such that α covers both α and β:

$$(\mathcal{G}, R \cdot [\alpha \to E_1, \beta \to E_2])\{\alpha/\beta\} \triangleq (\mathcal{G}\{\alpha/\beta\}, R\{\alpha/\beta\} \cdot [\alpha \to (E_1 \cup E_2)\{\alpha/\beta\}]).$$

[3] The simplify subroutine is similar to the widening operator in [4].

For example, consider the following transitions:

$$\begin{array}{l}
\alpha ::= \mathrm{nil} \mid \langle \beta, \beta \rangle \mid \langle \gamma, \gamma \rangle \,, \beta ::= \langle \gamma, \gamma \rangle \,, \gamma ::= \langle \mathrm{nil}, \mathrm{nil} \rangle \\
\overset{(\mathbf{case})}{\rightsquigarrow} \alpha ::= \mathrm{nil} \mid \langle \beta, \beta \rangle \,, \beta ::= \langle \beta, \beta \rangle \mid \langle \mathrm{nil}, \mathrm{nil} \rangle \overset{(\mathbf{nil})}{\rightsquigarrow} \alpha ::= \mathrm{nil} \mid \langle \beta, \beta \rangle \,, \beta ::= \langle \beta, \beta \rangle \mid \langle \beta, \mathrm{nil} \rangle \mid \mathrm{nil} \\
\overset{(\mathbf{nil})}{\rightsquigarrow} \alpha ::= \mathrm{nil} \mid \langle \beta, \beta \rangle \,, \beta ::= \langle \beta, \beta \rangle \mid \mathrm{nil} \qquad \overset{(\mathbf{def})}{\rightsquigarrow} \alpha ::= \mathrm{nil} \mid \langle \alpha, \alpha \rangle
\end{array}$$

In the initial grammar, α's definition has the similar cases $\langle \beta, \beta \rangle$ and $\langle \gamma, \gamma \rangle$, so we apply $\{\beta/\gamma\}$ (**case**). In the second grammar, β's definition has similar cases $\langle \beta, \beta \rangle$ and $\langle \mathrm{nil}, \mathrm{nil} \rangle$. Thus, we replace nil by β, and add the nil case to β's definition (**nil**). We apply (**nil**) once more for the second field. In the fourth grammar, since α and β have similar definitions, we apply $\{\alpha/\beta\}$ (**def**). As a result, we obtain the last grammar which says that α describes binary trees.

The last subroutine bound^k checks the number of symbolic locations in each shape graph. The subroutine bound^k simply gives $\top$ when one of shape graphs has more than k symbolic locations, thereby ensuring the termination of the analysis.[4]

$\mathsf{bound}^k(\mathcal{G}, R) = \text{if (no } (s, g) \text{ in } \mathcal{G} \text{ has more than } k \text{ symbolic locations) then } (\mathcal{G}, R) \text{ else } \top$

Lemma 1. *Given every abstract state* $(\mathcal{G}, R)$, $\mathsf{normalize}(\mathcal{G}, R)$ *always terminates, and its result is a* k*-bounded normalized abstract state.*

6 Analysis

Our analyzer (defined in Figure 3) consists of two parts: the "forward analysis" of commands C, and the "backward analysis" of boolean expressions B. Both of these interpret C and B as functions on abstract states, and they accomplish the usual goals in the static analysis: for an initial abstract state $(\mathcal{G}, R)$, $[\![C]\!](\mathcal{G}, R)$ approximates the possible output states, and $[\![B]\!](\mathcal{G}, R)$ denotes the result of pruning some states in $(\mathcal{G}, R)$ that do not satisfy B.

One particular feature of our analysis is that the analysis also checks the absence of memory errors, such as null-pointer dereference errors. Given a command C and an abstraction $(\mathcal{G}, R)$ for input states, the result $[\![C]\!](\mathcal{G}, R)$ of analyzing the command C can be either some abstract state $(\mathcal{G}', R')$ or $\top$. $(\mathcal{G}', R')$ means that all the results of C from $(\mathcal{G}, R)$ are approximated by $(\mathcal{G}', R')$, but in addition to this, it also means that no computations of C from $(\mathcal{G}, R)$ can generate memory errors. $\top$, on the other hand, expresses the possibility of memory errors, or indicates that a program uses the data structures whose complexity goes beyond the current capability of the analysis.

The analyzer unfolds the grammar definition by calling the subroutine unfold. Given a shape graph (s, g), a variable x and a grammar R, the subroutine unfold

[4] Limiting the number of symbolic locations to be at most k ensures the termination of the analyzer in the *worst case*. When programs use data structures that our grammar captures well, the analysis usually terminates without using this k limitation, and yields meaningful results.

$\boxed{[\![C]\!] : \widehat{D} \to \widehat{D}}$

$$
\begin{array}{rl}
[\![x := \texttt{new}]\!]\,(\mathcal{G},R) &= (\{(s[x{\to}a],\, g[a{\to}\langle b,c\rangle, b{\to}\mathrm{nil}, c{\to}\mathrm{nil}]) \mid (s,g)\in\mathcal{G}\},R) \text{ new } a,b,c\\
[\![x := \texttt{nil}]\!]\,(\mathcal{G},R) &= (\{(s[x{\to}a], g[a{\to}\mathrm{nil}]) \mid (s,g)\in\mathcal{G}\},R) \text{ new } a\\
[\![x := y]\!]\,(\mathcal{G},R) &= \text{when } y\in\mathrm{dom}(s) \text{ for all } (s,g)\in\mathcal{G},\\
&\quad (\{(s[x{\to}s(y)], g) \mid (s,g)\in\mathcal{G}\},R)\\
[\![x \texttt{->0} := y]\!]\,(\mathcal{G},R) &= \text{when } \mathsf{unfold}(\mathcal{G},R,x)=\mathcal{G}' \text{ and } \forall(s,g)\in\mathcal{G}'.\, y\in\mathrm{dom}(s),\\
&\quad (\{(s, g[s(x){\to}\langle s(y),c\rangle] \mid (s,g)\in\mathcal{G}',\ g(s(x))=\langle b,c\rangle\},R)\\
[\![x := y\texttt{->0}]\!]\,(\mathcal{G},R) &= \text{when } \mathsf{unfold}(\mathcal{G},R,y)=\mathcal{G}',\\
&\quad (\{(s[x{\to}b], g) \mid (s,g)\in\mathcal{G}',\ g(s(y))=\langle b,c\rangle\},R)\\
[\![C_1\,\texttt{;}\,C_2]\!]\,(\mathcal{G},R) &= [\![C_2]\!]\,([\![C_1]\!]\,(\mathcal{G},R))\\
[\![\texttt{if } B\ C_1\ C_2]\!]\,(\mathcal{G},R) &= [\![C_1]\!]\,([\![B]\!]\,(\mathcal{G},R)) \dot{\sqcup}\, [\![C_2]\!]\,([\![\texttt{!}B]\!]\,(\mathcal{G},R))\\
[\![\texttt{while } B\ C]\!]\,(\mathcal{G},R) &= [\![\texttt{!}B]\!]\left(\mathrm{lfix}^{\dot{\sqsubseteq}}\ \lambda A\colon \widehat{D}_k^{\nabla}.\ \mathsf{normalize}(A \dot{\sqcup} (\mathcal{G},R) \dot{\sqcup} [\![C]\!]\,([\![B]\!]A))\right)\\
[\![C]\!]A &= \top \text{ (other cases)}
\end{array}
$$

$\boxed{[\![B]\!] : \widehat{D} \to \widehat{D}}$

$$
\begin{array}{rl}
[\![x = y]\!]\,(\mathcal{G},R) &= \text{when } \mathsf{split}(\mathsf{split}((\mathcal{G},R),x),y) = (\mathcal{G}',R')\\
&\quad (\{(s,g)\in\mathcal{G}' \mid s(x){\equiv}s(y) \vee g(s(x)){=}g(s(y)){=}\mathrm{nil})\},\ R')\\
[\![\texttt{!}x = y]\!]\,(\mathcal{G},R) &= \text{when } \mathsf{split}(\mathsf{split}((\mathcal{G},R),x),y) = (\mathcal{G}',R')\\
&\quad (\{(s,g)\in\mathcal{G}' \mid s(x){\not\equiv}s(y) \wedge (g(s(x)){\neq}\mathrm{nil} \vee g(s(y)){\neq}\mathrm{nil})\},\ R')\\
[\![\texttt{!(!}B\texttt{)}]\!]\,(\mathcal{G},R) &= [\![B]\!]\,(\mathcal{G},R)\\
[\![B]\!]\,A &= \top \text{ (other cases)}
\end{array}
$$

Subroutine unfold unrolls the definition of a grammar:

$$
\mathsf{unfold}((s,g),R,x) = \begin{cases} \{(s, g[a{\to}\langle b,c\rangle, b{\to}v, c{\to}u])\mid \langle v,u\rangle{\in}R(a)\} & \text{if } g(s(x)){=}\alpha \wedge \mathrm{nil}{\notin}R(\alpha)\\ \{(s,g)\} & \text{if } g(s(x)) \text{ is a pair}\\ \top & \text{otherwise}\end{cases}
$$

$$
\mathsf{unfold}(\mathcal{G},R,x) = \begin{cases} \bigcup_{(s,g)\in\mathcal{G}} \mathsf{unfold}((s,g),R,x) & \text{if } \forall(s,g)\in\mathcal{G}.\ \mathsf{unfold}((s,g),R,x)\neq\top\\ \top & \text{otherwise}\end{cases}
$$

Subroutine $\mathsf{split}((s,g),R,x)$ changes (s,g) to (s',g') s.t. $s'(x)$ means nil iff $g'(s'(x))$=nil.

$$
\begin{array}{l}
\mathsf{split}((s,g),R,x) = \text{if } (\exists\alpha.\, g(s(x)){=}\alpha \wedge R(\alpha){\supseteq}\{\mathrm{nil}\} \wedge R(\alpha)\neq\{\mathrm{nil}\})\\
\qquad \text{then } (\{(s,g[s(x){\to}\mathrm{nil}]), (s,g[s(x){\to}\beta])\}, R[\beta{\to}R(\alpha){-}\{\mathrm{nil}\}]) \text{ for fresh } \beta\\
\qquad \text{else if } (\exists\alpha.\, g(s(x)){=}\alpha \wedge R(\alpha){=}\{\mathrm{nil}\}) \text{ then } (\{(s,g[s(x){\to}\mathrm{nil}])\},R)\\
\qquad\qquad \text{else } (\{(s,g)\},R)
\end{array}
$$

$$
\mathsf{split}(\mathcal{G},R,x) = \begin{cases} \dot{\bigcup}_{(s,g)\in\mathcal{G}} \mathsf{split}((s,g),R,x) & \text{if } \forall(s,g){\in}\mathcal{G}.\, x\in\mathrm{dom}(s)\\ \top & \text{otherwise}\end{cases}
$$

The algorithmic order $\dot{\sqsubseteq}$ defined in [8] satisfies that if $A \dot{\sqsubseteq} B$, $\mathsf{means}(A) \Rightarrow \mathsf{means}(B)$

Fig. 3. Analysis

first checks whether $g(s(x))$ is a nonterminal or not. If $g(s(x))$ is a nonterminal α, unfold looks up the definition of α in R and unrolls this definition in the shape graph (s,g): for each case e in $R(\alpha)$, it updates g by $[s(x){\to}e]$. For instance, when $R(\beta) = \{\langle\beta,\gamma\rangle, \langle\delta,\delta\rangle\}$, $\mathsf{unfold}(([x{\to}a],[a{\to}\beta]),R,x)$ is shape-graph set $\{([x{\to}a],[a{\to}\langle\beta,\gamma\rangle]),\ ([x{\to}a],[a{\to}\langle\delta,\delta\rangle])\}$.

7 Full Analysis

The basic version of our analysis, which we have presented so far, cannot deal with data structures with sharing, such as doubly linked lists and binomial heaps. If a program uses such data structures, the analysis gives up and returns $\top$.

The full analysis overcomes this shortcoming by using a more expressive language for a grammar, where a nonterminal is allowed to have parameters. The main feature of this new parameterized grammar is that an invariant for a data structure with sharing is expressible by a grammar, as long as the sharing is "cyclic." A parameter plays a role of "targets" of such cycles.

The overall structure of the full analysis is almost identical to the basic version in Figure 3. Only the subroutines, such as $\mathsf{normalize}$, are modified. In this section, we will explain the full analysis by focusing on the new parameterized grammar, and the modified normalization function for this grammar. The full definition is in [8].

7.1 Abstract Domain

Let self and arg be two different symbolic locations. In the full analysis, the domains for shape graphs and grammars are modified as follows:

$$NTermApp \triangleq NonTerm \times (SymLoc + \bot) \quad NTermAppR \triangleq NonTerm \times (\{\mathsf{self}, \mathsf{arg}\} + \bot)$$
$$Graph \triangleq (Vars \rightharpoonup_{\mathrm{fin}} SymLoc) \times (SymLoc \rightharpoonup_{\mathrm{fin}} \{\mathrm{nil}\} + SymLoc^2 + NTermApp)$$
$$Grammar \triangleq NonTerm \rightharpoonup_{\mathrm{fin}} \wp_{\mathrm{nf}}(\{\mathrm{nil}\} + (\{\mathrm{nil}\} + \{\mathsf{self}, \mathsf{arg}\} + NTermAppR)^2)$$

The main change in the new definitions is that all the nonterminals have parameters. All the uses of nonterminals in the old definitions are replaced by the applications of nonterminals, and the declarations of nonterminals in a grammar can use two symbolic locations self and arg, as opposed to none, which denote the implicit self parameter and the explicit parameter.[5] For instance, a doubly-linked list is defined by $\mathsf{dll} ::= \mathrm{nil} \mid \langle \mathsf{arg}, \mathsf{dll}(\mathsf{self}) \rangle$. This grammar maintains the invariant that arg points to the previous cell. So, the first field of a node always points to the previous cell, and the second field the the next cell. Note that $\bot$ can be applied to a nonterminal; this means that we consider subcases of the nonterminal where the arg parameter is not used. For instance, if a grammar R maps β to $\{\mathrm{nil}, \langle \mathsf{arg}, \mathsf{arg} \rangle\}$, then $\beta(\bot)$ excludes $\langle \mathsf{arg}, \mathsf{arg} \rangle$, and means the empty heap object.

As in the basic case, the precise meaning of a shape graph and a grammar is given by a translation into separation-logic assertions. We can define a translation means by modifying only $\mathsf{means}_{\mathrm{v}}$ and $\mathsf{means}_{\mathrm{g}}$.

$$\mathsf{means}_{\mathrm{v}}(a, \mathrm{nil}) \triangleq a \doteq \mathrm{nil} \quad \mathsf{means}_{\mathrm{v}}(a, \alpha(b)) \triangleq \alpha(a, b)$$
$$\mathsf{means}_{\mathrm{v}}(a, b) \triangleq a \doteq b \quad \mathsf{means}_{\mathrm{v}}(a, \alpha(\bot)) \triangleq \forall b.\alpha(a, b)$$

In the last clause, b is a different variable from a. The meaning of a grammar is a context defining a set of recursive predicates.

[5] We allow only "one" explicit parameter. So, we can use pre-defined name arg.

$$\begin{aligned}
\mathsf{means}_{\mathrm{g}}(R) &\triangleq \{\alpha(a,b) = \bigvee_{e \in R(\alpha)} \mathsf{means}_{\mathrm{gc}}(a,b,e)\}_{\alpha \in \mathrm{dom}(R)} \\
\mathsf{means}_{\mathrm{gc}}(a,b,\mathsf{nil}) &\triangleq \mathsf{means}_{\mathrm{v}}(a,\mathsf{nil}) \\
\mathsf{means}_{\mathrm{gc}}(a,b,\langle v_1,v_2\rangle) &\triangleq \exists a_1 a_2.\, (a \mapsto a_1,a_2) * \mathsf{means}_{\mathrm{v}}(a_1, v_1\,\{a/\mathsf{self}, b/\mathsf{arg}\}) \\
&\qquad\qquad * \mathsf{means}_{\mathrm{v}}(a_2, v_2\,\{a/\mathsf{self}, b/\mathsf{arg}\})
\end{aligned}$$

In the second clause, a_1 and a_2 are variables that do not appear in v_1, v_2, a, b.

7.2 Normalization Function

To fully exploit the increased expressivity of the abstract domain, we change the normalization function in the full analysis. The most important change in the new normalization function is the addition of new rules (**cut**) and (**bfold**) into the fold procedure.

The (**cut**) rule enables the conversion of a cyclic structure to grammar definitions. Recall that the (**fold**) rule can recognize a heap object only when the object does not have shared cells internally. The key idea is to "cut" a "non-critical" link to a shared cell, and represent the removed link by a parameter to a nonterminal. If enough such links are cut from an heap object, the object no longer has (explicitly) shared cells, so that the wrapping step of (**fold**) can be applied. The formal definition of the (**cut**) rule is:

$$(\mathbf{cut})\ (\mathcal{G} \uplus \{(s, g \cdot [a \to \langle a_1, a_2\rangle])\}, R) \rightsquigarrow \begin{pmatrix} \mathcal{G} \cup \{(s, g \cdot [a \to \alpha(b)])\}, \\ R \cdot [\alpha \to \{\langle a_1, a_2\rangle\,\{\mathsf{self}/a, \mathsf{arg}/b\}\}] \end{pmatrix}$$

where there are paths from variables to a_1 and a_2 in g, $\mathsf{free}(\langle v_1, v_2\rangle) \subseteq \{a, b\}$, and α is fresh. (If $\mathsf{free}(\langle v_1, v_2\rangle) \subseteq \{a\}$, we use $\alpha(\bot)$ instead of $\alpha(b)$.)

Figure 4.(a) shows how a cyclic structure is converted to grammar definitions.[6] In the first shape graph, "cell" a is shared because variable x points to a and "cell" c points to a, but the link from c to a is not critical because even without this link, a is still reachable from x. Thus, the (**cut**) rule cuts the link from c to a, introduces a nonterminal α_c with the definition $\{\langle \mathsf{arg}\rangle\}$, and annotates node c with $\alpha_c(a)$. Note that the resulting graph (the second shape graph in Figure 4.(a)) does not have explicit sharing. So, we can apply the (**fold**) rule to c, and then to b as shown in the last two shape graphs in Figure 4.(a).

The (**bfold**) rule wraps a cell "from the back." Recall that the (**fold**) rule puts a cell at the front of a heap object; it adds the cell as a root for a nonterminal. The (**bfold**) rule, on the other hand, puts a cell a at the exit of a heap object. When b is used as a parameter for a nonterminal α, the rule "combines" b and α. This rule can best be explained using a list-traversing algorithm. Consider a program that traverses a linked list, where variable r points to the head cell of the list, and variable c to the current cell of the list. The usual loop invariant of such a program is expressed by the first shape graph in Figure 4.(b). However, only with the (**fold**) rule, which adds a cell to the front, we cannot discover this invariant; one iteration of the program moves c to the next cell, and thus changes the shape graph into the second shape graph in Figure 4.(b), but this

[6] To simplify the presentation, we assume that each cell in the figure has only a single field.

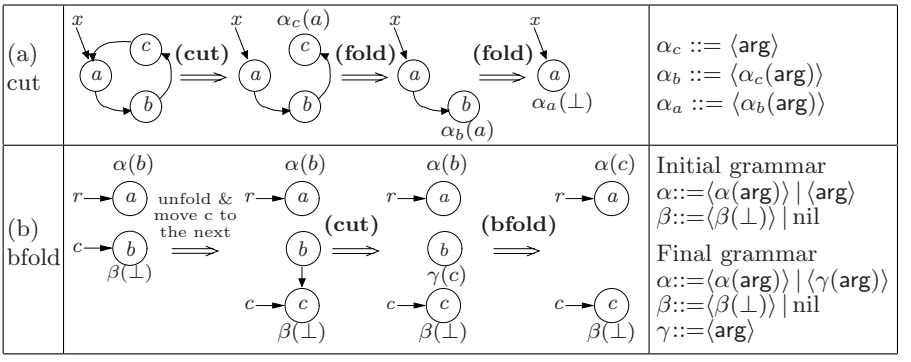


Fig. 4. Examples of (**cut**) and (**bfold**)

new graph is not similar to the initial one. The (**bfold**) rule changes the new shape graph back to the one for the invariant, by merging $\alpha(b)$ with cell b. The (**cut**) rule first cuts the link from b to c, extends a grammar with $[\gamma \to \{\langle \mathsf{arg} \rangle\}]$, and annotates the node b with $\gamma(c)$. Then, the (**bfold**) rule finds all the places where arg is used as itself in the definition of α, and replaces arg there by $\gamma(\mathsf{arg})$. Finally, the rule changes the binding for a from $\alpha(b)$ to $\alpha(c)$, and eliminates cell b, thus resulting the last shape graph in Figure 4(b).[7] The precise definition of (**bfold**) does what we call *linearity* check, in order to ensure the soundness of replacing arg by nonterminals:[8]

(**bfold**) $(\mathcal{G} \cup \{(s, g \cdot [a \to \alpha(b), b \to \beta(w)])\}, R) \rightsquigarrow (\mathcal{G} \cup \{(s, g \cdot [a \to \alpha'(w)])\}, R \cdot [\alpha' \to E])$
where b does not appear in g, α is linear (that is, arg appears exactly once in each case of $R(\alpha)$), and $E = \{\mathsf{nil} \in R(\alpha)\} \cup \{\langle f(v_1), f(v_2) \rangle \mid \langle v_1, v_2 \rangle \in R(\alpha)\}$
where $f(v) = \mathsf{if}\ (v \equiv \mathsf{arg})\ \mathsf{then}\ \beta(\mathsf{arg})\ \mathsf{else}\ \Big(\mathsf{if}\ (v \equiv \alpha(\mathsf{arg}))\ \mathsf{then}\ \alpha'(\mathsf{arg})\ \mathsf{else}\ v\Big)$.

7.3 Correctness

The correctness of our analysis is expressed by the following theorem:

Theorem 1. *For all programs C and abstract states $(\mathcal{G}, R)$, if $[\![C]\!](\mathcal{G}, R)$ is a non-$\top$ abstract state $(\mathcal{G}', R')$, then triple $\{\mathsf{means}(\mathcal{G}, R)\}C\{\mathsf{means}(\mathcal{G}', R')\}$ holds in separation logic.*

We proved this theorem in two steps. First, we showed a lemma that all subroutines, such as normalize and unfold, and the backward analysis are correct. Then,

[7] The grammar is slightly different from the one for the invariant. However, if we combine two abstract states and apply unify and simplify, then the grammar for the invariant is recovered.

[8] Here we present only for the case that the parameter of α is not passed to another different nonterminals. With such nonterminals, we need to do a more serious linearity check on those nonterminals, before modifying the grammar.

Table 1. Experimental Results

program	description	cost(sec)	analysis result
`listrev.c`	list construction followed by list reversal	0.01	the result is a list
`dbinary.c`	construction of a tree with parent pointers	0.01	the result is a tree with parent pointers
`dll.c`	doubly-linked list construction	0.01	the result is a doubly-linked list
`bh.c`	binomial heap construction	0.14	the result is a binomial heap
`sw.c`	Schorr-Waite tree traversal	0.05	the result is a tree
`swfree.c`	Schorr-Waite tree disposal	0.02	the tree is completely disposed

For all the examples, our analyzer proves the absence of null pointer dereference errors and memory leaks.

with this lemma, we applied the induction on the structure of C, and showed that $\{\mathsf{means}(\mathcal{G}, R)\}C\{\mathsf{means}(\mathcal{G}', R')\}$ is derivable in separation logic. The validity of the triple now follows, because separation-logic proof rules are sound. The details are in [8].

8 Experiments

We have tested our analysis with the six programs in Table 1. For each of the programs, we ran the analyzer, and obtained abstract states for a loop invariant and the result. In this section, we will explain the cases of binomial heap construction and the Schorr-Waite tree traversal. The others are explained at `http://ropas.snu.ac.kr/grammar`.

Binomial Heap Construction. In this experiment, we took an implementation of binomial heap construction in [1], where each cell has three pointers: one to the left-most child, another to the next sibling, and the third to the parent. We ran the analyzer with this binomial heap construction program and the empty abstract state $(\{\}, [])$. Then, the analyzer inferred the following same abstract state $(\mathcal{G}, R)$ for the result of the construction as well as for the loop invariant. Here we omit $\bot$ from $\mathsf{forest}(\bot)$.

$$\mathcal{G} = \{([x{\rightarrow}a], [a{\rightarrow}\mathsf{forest}])\} \quad R = \begin{bmatrix} \mathsf{forest} ::= \mathrm{nil} \mid \langle \mathsf{stree}(\mathsf{self}), \mathsf{forest}, \mathrm{nil} \rangle \,, \\ \mathsf{stree} ::= \mathrm{nil} \mid \langle \mathsf{stree}(\mathsf{self}), \mathsf{stree}(\mathsf{arg}), \mathsf{arg} \rangle \end{bmatrix}$$

The unique shape graph in $\mathcal{G}$ means that the heap has only a single heap object whose root is stored in x, and the heap object is an instance of forest. Grammar R defines the structure of this heap object. It says that the heap object is a linked list of instances of stree, and that each instance of stree in the list is given the address of the containing list cell. These instances of stree are, indeed, precisely those trees with pointers to the left-most children and to the next sibling, and the parent pointer.

Schorr-Waite Tree Traversal. We used the following $(\mathcal{G}_0, R_0)$ as an initial abstract state:

$$\mathcal{G}_0 = \{([x{\rightarrow}a], [a{\rightarrow}\mathsf{tree}])\} \quad R_0 = [\mathsf{tree} ::= \mathrm{nil} \mid \langle \mathrm{I}, \mathsf{tree}, \mathsf{tree}\rangle]$$

Here we omit $\perp$ from $\mathsf{tree}(\perp)$. This abstract state means that the initial heap contains only a binary tree a whose cells are marked I.

Given the traversing algorithm and the abstract state $(\mathcal{G}_0, R_0)$, the analyzer produced $(\mathcal{G}_1, R_1)$ for final states, and $(\mathcal{G}_2, R_2)$ for a loop invariant:

$$\mathcal{G}_1 = \{([x{\rightarrow}a], [a{\rightarrow}\mathsf{treeR}])\} \qquad R_1 = [\mathsf{treeR} ::= \mathrm{nil} \mid \langle \mathrm{R}, \mathsf{treeR}, \mathsf{treeR}\rangle]$$

$$\mathcal{G}_2 = \{([x{\rightarrow}a, y{\rightarrow}b], [a{\rightarrow}\mathsf{treeRI}, b{\rightarrow}\mathsf{rtree}])\}$$

$$R_2 = \begin{bmatrix} \mathsf{rtree} ::= \mathrm{nil} \mid \langle \mathrm{R}, \mathsf{treeR}, \mathsf{rtree}\rangle \mid \langle \mathrm{L}, \mathsf{rtree}, \mathsf{tree}\rangle\,, & \mathsf{tree} ::= \mathrm{nil} \mid \langle \mathrm{I}, \mathsf{tree}, \mathsf{tree}\rangle\,, \\ \mathsf{treeR} ::= \mathrm{nil} \mid \langle \mathrm{R}, \mathsf{treeR}, \mathsf{treeR}\rangle\,, & \mathsf{treeRI} ::= \mathrm{nil} \mid \langle \mathrm{I}, \mathsf{tree}, \mathsf{tree}\rangle \mid \langle \mathrm{R}, \mathsf{treeR}, \mathsf{treeR}\rangle \end{bmatrix}$$

The abstract state $(\mathcal{G}_1, R_1)$ means that the heap contains only a single heap object x, and that this heap object is a binary tree containing only R-marked cells. Note that this abstract state implies the absence of memory leaks because the tree x is the only thing in the heap.

The loop invariant $(\mathcal{G}_2, R_2)$ means that the heap contains two disjoint heap objects x and y. Since the heap object x is an instance of treeRI, the object x is an I-marked binary tree, or an R-marked binary tree. This first case indicates that x is first visited, and the second case that x has been visited before. The nonterminal rtree for the other heap object y implies that one of left or right field of cell y is reversed. The second case, $\langle \mathrm{R}, \mathsf{treeR}, \mathsf{rtree}\rangle$, in the definition of rtree means that the current cell is marked R, its right field is reversed, and the left subtree is an R-marked binary tree. The third case, $\langle \mathrm{L}, \mathsf{rtree}, \mathsf{tree}\rangle$, means that the current cell is marked L, the left field is reversed, and the right subtree is an I-marked binary tree. Note that this invariant, indeed, holds because y points to the parent of x, so the left or right field of cell y must be reversed.

References

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, 2001.
2. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.
3. Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Logic and Comput.*, 2(4):511–547, 1992.
4. Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 170–181, La Jolla, California, June 1995. ACM Press, New York, NY.
5. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 230–241. ACM Press, 1994.

6. Pascal Fradet and Daniel Le Métayer. Shape types. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 27–39. ACM Press, January 1997.
7. Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1993.
8. Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Automatic verification of pointer programs using grammar-based shape analysis. Tech. Memo. ROPAS-2005-23, Programming Research Laboratory, School of Computer Science & Engineering, Seoul National University, March 2005.
9. R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *Proceedings of the International Symposium on Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 265–279, August 2004.
10. A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. ACM, June 2001.
11. Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 268–280. ACM Press, January 2004.
12. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, July 2002.
13. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, January 1998.
14. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
15. Élodie-Jane Sims. Extending separation logic with fixpoints and postponed substitution. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology*, volume 3116 of *Lecture Notes in Computer Science*, pages 475–490, 2004.

A Type Discipline for Authorization Policies

Cédric Fournet[1], Andrew D. Gordon[1], and Sergio Maffeis[1,2]

[1] Microsoft Research
[2] Department of Computing, Imperial College London

Abstract. Distributed systems and applications are often expected to enforce high-level authorization policies. To this end, the code for these systems relies on lower-level security mechanisms such as, for instance, digital signatures, local ACLs, and encrypted communications. In principle, authorization specifications can be separated from code and carefully audited. Logic programs, in particular, can express policies in a simple, abstract manner.

For a given authorization policy, we consider the problem of checking whether a cryptographic implementation complies with the policy. We formalize authorization policies by embedding logical predicates and queries within a spi calculus. This embedding is new, simple, and general; it allows us to treat logic programs as specifications of code using secure channels, cryptography, or a combination. Moreover, we propose a new dependent type system for verifying such implementations against their policies. Using Datalog as an authorization logic, we show how to type several examples using policies and present a general schema for compiling policies.

1 Typing Implementations of Authorization Policies

An *authorization policy* prescribes conditions that must be satisfied before performing any privileged action (for example, accessing a sensitive resource). A system complies with the policy if these conditions hold whenever the action is performed—however, the policy does not usually prescribe a particular choice of enforcement mechanisms.

Authorization issues can be complex, even at an abstract level. Some policies address security concerns for numerous actors, involving roles, groups, partial trust, and controlled delegation. Those policies are best expressed in high-level languages, with supporting tools. Specifically, logic programming seems well suited for expressing policies: each authorization request is formulated as a logical request against a database of facts and rules, while the policy itself carefully controls changes to the database. Hence, variants of Datalog have been usefully applied to design trust management systems (e.g., PolicyMaker [6], SD3 [20], Binder [12]), express complex policies (e.g., Cassandra [4]), and study authorization languages (e.g., SDSI/SPKI [1, 21], XrML [11]).

Given a target policy, we consider the problem of verifying that a particular system correctly implements this policy. In a distributed setting, this refinement typically involves security protocols and cryptography. For instance, when receiving a request, one may first verify an identity certificate, then authenticate the message, and finally consider the privileges associated with the sender. Authorization decisions are often intermingled with other imperative code, and are hard to analyze and audit. For instance,

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 141–156, 2005.

the request may rightfully appear in many places in the code, most of them without a valid identity certificate at hand. The relation between imperative code and declarative policies is usually informal: theoretical studies rarely connect the logic to an operational semantics.

Our formal development is within a spi calculus [3], that is, a pi calculus with abstract cryptographic operations. We use a global policy, interpreted against processes in a way that generalizes a previous embedding [17] of correspondence assertions [24]. There are many techniques to verify standard correspondences with respect to the Dolev-Yao model [13], the standard "network is the opponent" threat model for cryptographic protocols. However, these correspondences are attached to low-level events (such as a successful decryption), and it can be quite hard to relate them to high-level access control decisions. Perhaps in consequence, more abstract correspondences have seldom been validated against the Dolev-Yao model, even though they rely on cryptography.

In contrast to several previous works, we use the authorization language as a statically enforced specification, instead of a language for programming dynamic authorization decisions. The two approaches are complementary. The static approach is less expressive in terms of policies, as we need to anticipate the usage of the facts and rules involved at runtime. In contrast, a logic-based implementation may dynamically accept (authenticated) facts and rules, as long as they lead to a successful policy evaluation. The static approach is more expressive in terms of implementations, as we can assemble imperative and cryptographic mechanisms (for example, communications to collect remote certificates), irrespective of the logic-based evaluation strategy suggested by the policy. Hence, the static approach may be more efficient and pragmatically simpler to adapt to existing systems. Non-executable policies may also be simpler to write and to maintain, as they can safely ignore functional issues.

Summary of Contributions. To our knowledge, this is the first attempt to relate authorization logics to their cryptographic implementation in a pi calculus. Specifically:

- We show how to embed a range of authorization logics within a pi calculus. (We use Datalog as a simple, concrete example of an authorization logic.)
- We develop a new type system that checks conformance to a logic policy by keeping track of logical facts and rules in the typing environment, and using logical deduction to type authorization queries. Our main theorem states that all queries activated in a well-typed program follow from the enclosing policy.
- As a sample application, we present two distributed implementations of a simple Datalog policy for conference management featuring rules for filing reports and delegating reviews. One implementation requests each delegation to be registered online, whereas the other enables offline, signature-based delegation, and checks the whole delegation chain later, when a report is filed.
- As another, more theoretical application, we present a generic implementation of Datalog in the pi calculus—well-typed in our system—which can be used as a default centralized implementation for any part of a policy.

We built a typechecker and a symbolic interpreter for our language, and used them to validate these applications. Our initial experience confirms the utility of such tools for writing code that composes several protocols, even if its overall size remains modest so far (a few hundred lines).

Related Work. There is a substantial literature on type systems for checking security properties. In the context of process calculi, there are, for example type systems to check secrecy [2] and authenticity [16] properties in the spi calculus, access control properties of mobile code in the boxed ambient calculus [8], and discretionary access control [9] and role-based access control [7] in the pi calculus. Moreover, various experimental systems, such as JIF [22] and KLAIM [23], include types for access control. Still, there appears to be no prior work on typing implementations of a general authorization logic.

In the context of strand spaces and nonce-based protocols, Guttman *et al.* [19] annotate send actions in a protocol with trust logic formulas which must hold when a message is sent, and receive actions with formulas which can be assumed to hold when a message is received. Their approach also relies on logically-defined correspondence properties, but it assumes the dynamic invocation of an external authorization engine, thereby cleanly removing the dependency on a particular authorization policy when reasoning about protocols. More technically, we attach static authorization effects to any operation (input, decryption, matching) rather than just message inputs.

Blanchet's ProVerif [5] checks correspondence assertions in the applied pi calculus by reduction to a logic programming problem. ProVerif can check complex disjunctive correspondences, but has not been applied to check general authorization policies.

Guelev *e*t al. [18] also adopt a conference programme committee as a running example, in the context of model checking the consequences of access control policies.

Contents. The paper is organized as follows. Section 2 reviews Datalog, illustrates its usage to express authorization policies, and states a general definition of authorization logics. Section 3 defines a spi calculus with embedded authorization assertions. Section 4 presents our type system and states our main safety results. Section 5 develops well-typed distributed implementations for our sample delegation policy. Section 6 provides our pi calculus implementation of Datalog and states its correctness and completeness. Section 7 concludes and sketches future work. Due to space constraints, some standard definitions and all proofs are omitted; they appear in a technical report [14].

2 A Simple Logic for Authorization

Datalog. We briefly present a syntax and semantics for Datalog. (For a comprehensive survey of Datalog, see for instance [10].) A Datalog program consists of *facts*, which are statements about the universe of discourse, and *clauses*, which are rules that can be used to infer facts. In the following, we interpret programs as authorization policies.

Syntax for Datalog:

$u ::= X \mid M$	term: a logic variable X or a spi calculus message M
$L ::= p(u_1,\ldots,u_n)$	literal: predicate p holds for $u_1,\ldots,u_n$
$C ::= L :- L_1,\ldots,L_n$	clause (or rule), with $n \geq 0$ and $fv(L) \subseteq \bigcup_i fv(L_i)$
$S ::= \{C_1,\ldots,C_n\}$	Datalog program (or policy): a set of clauses

A literal L is a predicate $p(u_1,\ldots,u_n)$, of fixed arity $n \geq 0$, on terms $u_1,\ldots,u_n$. Terms range over logical variables X,Y,Z and messages M; these messages are treated as Datalog atoms, but they have some structure in our spi calculus, defined in Section 3.

A clause $L:-L_1,\ldots,L_n$ has a *head*, L, and a *body*, $L_1,\ldots,L_n$; it is intuitively read as the universal closure of the propositional formula $L_1\wedge\ldots\wedge L_n\rightarrow L$. In a clause, variables occurring in the body bind those occurring in the head. A phrase of syntax is *ground* if it has no free variables. We require that each clause be ground. A *fact* F is a clause with an empty body, $L:-$. We often write the (ground) literal L as an abbreviation of the fact $L:-$.

We use the following notations: for any phrase φ, we let $fn(\varphi)$ and $fv(\varphi)$ collect free spi calculus names and free variables, respectively. We write $\widetilde{\varphi}$ for the tuple $\varphi_1,\ldots,\varphi_t$, for some $t\geq 0$. We write $\{u/X\}$ for the capture-avoiding substitution of u for X, and write $\{\widetilde{u}/\widetilde{X}\}$ instead of $\{u_1/X_1\}\ldots\{u_n/X_n\}$. We let σ range over these substitutions. Similarly, we write $\{M/n\}$ for capture-avoiding substitution of message M for name n.

A fact can be derived from a Datalog program using the rule below:

Logical Inference: $S\models F$

(Infer Fact)

$$\frac{L:-L_1,\ldots,L_n\in S \qquad S\models L_i\sigma \quad \forall i\in 1..n}{S\models L\sigma} \quad \text{for } n\geq 0$$

More generally, a clause C is *entailed* by a program S, also written $S\models C$, when we have $\{F \mid S'\cup\{C\}\models F\}\subseteq\{F \mid S'\cup S\models F\}$ for all programs S'. Similarly, C is *uniformly contained in* S when the inclusion above holds for all programs S' containing only facts. Entailment is a contextual property for programs: if $S\models C$ and $S\subseteq S'$, then $S'\models C$. We rely on this property when we reason about partial programs. We generalize inference to clauses accordingly:

Logical Inference for Clauses (Entailment): $S\models C$

(Infer Clause)

$$\frac{S\cup\{L_1\sigma,\ldots,L_n\sigma\}\models L\sigma \qquad \sigma \text{ maps } fv(L_1,\ldots,L_n) \text{ to fresh, distinct atoms}}{S\models L:-L_1,\ldots,L_n}$$

Example. Our main example application is a simplified conference management system, in charge of assigning papers to referees and collecting their reports. For simplicity, we focus on the fragment of the policy that controls the right to file a paper report in the system, from the conference manager's viewpoint. This right, represented by the predicate Report(U,ID,R), is parameterized by the principal who wrote the report, a paper identifier, and the report content. It means that principal U can submit report R on paper ID. For instance, the fact Report(alice,42,report42) authorizes a single report to be filed. Preferably, such facts should be deducible from the policy, rather than added to the policy one at a time. To this end, we introduce a few other predicates.

Some predicates represent the content of some *extensional* database of explicitly given facts. In our example, for instance, PCMember(U) means that principal U is a member of the committee; Referee(U,ID) means that principal U has been asked to review ID; and Opinion(U,ID,R) means that principal U has written report R on paper ID. Other predicates are *intensional*; they represent views computed from this authorization database. For instance, one may decide to specify Report(U,ID,R) using two clauses:

Report(U,ID,R):−Referee(U,ID),Opinion(U,ID,R) (clause A)
Report(U,ID,R):−PCMember(U),Opinion(U,ID,R) (clause B)

These clauses state that U can report R on ID if she has this opinion and, moreover, either U has been assigned this paper (clause A), or U is in the programme committee (clause B)—thereby enabling PC members to file reports on any paper even if it has not been assigned to them. Variants of this policy are easily expressible; for instance, we may instead state that PC members can file only subsequent reports, not initial ones, by using a recursive variant of clause B:

$$\text{Report(U,ID,R)} :- \text{PCMember(U)},\text{Opinion(U,ID,R)},\text{Report(V,ID,S)}$$

Delegation. Continuing with our example, we extend the policy to enable any designated referees to delegate their task to a subreferee. To this end, we add an extensional predicate, Delegate(U,V,ID), meaning that principal U intends to delegate paper ID to principal V, and we add a clause to derive new facts Referee(V,ID) accordingly:

$$\text{Referee(V,ID)} :- \text{Referee(U,ID)},\text{Delegate(U,V,ID)} \qquad \text{(clause C)}$$

Conversely, the policy { A,B,C } does not enable a PC member to delegate a paper, unless the paper has been assigned to her.

Discussion. In contrast to more sophisticated authorization languages, which associates facts with principals "saying" them, we adopt the subjective viewpoint of the conference system, which implicitly owns all predicates used to control reports. Even if Opinion(U,_) and Delegate(U,...) are implicitly owned by U, these predicates represent the fact that the conference system believes these facts, rather than U's intents. Also, the distinction between intensional and extensional predicates is useful to interpret policies but is not essential. As we illustrate in Section 5, this distinction in the specification does not prescribe any implementation strategy.

From Datalog to Arbitrary Authorization Logics. Although Datalog suffices as an authorization logic for the examples and applications developed in this paper, its syntax and semantics are largely irrelevant to our technical developments. More abstractly, our main results hold for any logic that meets the requirements listed below:

Definition 1. *An* authorization logic $(\mathscr{C}, fn, \models)$ *is a set of clauses* $C \in \mathscr{C}$ *closed by substitutions* σ *of messages for names, with finite sets of* free names $fn(C)$ *such that* $C\sigma = C$ *if* $dom(\sigma) \cap fn(C) = \varnothing$ *and* $fn(C\sigma) \subseteq (fn(C) \setminus dom(\sigma)) \cup fn(\sigma)$*; and with an* entailment relation $S \models C$*, between sets of clauses* $S \subseteq \mathscr{C}$ *and clauses* $C, C' \in \mathscr{C}$*, such that* (Mon) $S \models C \Rightarrow S \cup \{C'\} \models C$ *and* (Subst) $S \models C \Rightarrow S\sigma \models C\sigma$.

3 A Spi Calculus with Authorization Assertions

The spi calculus [3] extends the pi calculus with abstract cryptographic operations in the style of Dolev and Yao [13]. Names represent both cryptographic keys and communication channels. The version of spi given here has a small but expressive range of primitives: encryption and decryption using shared keys, input and output on shared channel names, and operations on pairs. We conjecture our results, including our type system, would smoothly extend to deal with more complex features such as asymmetric cryptography and communications, and a richer set of data types.

The main new features of our calculus are authorization assertions: statements and expectations. These processes generalize the begin- and end-assertions in previous embeddings of correspondences in process calculi [17]. Similarly, they track security properties, but do not in themselves affect the behaviour of processes.

A *statement* is simply a clause C (either a fact or a rule). For example, the following process is a composition of clause A of Section 2 with two facts:

A | Referee(alice,42) | Opinion(alice,42,report42) (process P)

An *expectation* **expect** C represents the expectation on the part of the programmer that the rule or fact C can be inferred from clauses in parallel. Expectations typically record authorization conditions. For example, the following process represents the (justified) expectation that a certain fact follows from the clauses of P.

P | **expect** Report(alice,42,report42) (process Q)

Expectations most usefully concern variables instantiated at runtime. In the following, the contents x of the report is received from the channel c:

P | **out** c (report42,**ok**) | **in** c(x,y); **expect** Report(alice,42,x) (process R)

(The distinguished name **ok** is an annotation to help typing, with no effect at runtime.)

All the statements arising in our case studies fall into two distinct classes. One class consists of unguarded, top-level statements of authorization rules, such as those in the previous section, that define the global authorization policy. The other class consists of input-guarded statements, triggered at runtime, that declare facts—not rules—about data arising at runtime, such as the identities of particular reviewers or the contents of reports. Moreover, all the expectations in our case studies are of facts, not rules.

The syntax and the operational semantics of our full calculus appear on the next page. The **split** and **match** processes for destructing pairs are worth comparing. A **split** binds names to the two parts of a pair, while a **match** is effectively a **split** followed by a conditional; think of **match** M **as** $(N,y);P$ as **split** M **as** (x,y);**if** $x = N$ **then** P. Taking **match** as primitive is a device to avoid using unification in a dependent type system [16]. Binding occurrences of names have type annotations, T or U; the syntax of our system of dependent types is in Section 4.

The operational semantics is defined as a reduction relation, with standard rules. Statements and expectations are inert processes; they do not have particular rules for reduction or congruence (although they are affected by other rules). The conditional operations **decrypt**, **split**, and **match** simply get stuck if decryption or matching fails; we could allow alternative branches for error handling, but they are not needed for the examples in the paper.

In examples, we rely on derived notations for n-ary tuples, and for pattern-matching tuples via sequences of match and split operations. For $n > 2$, $(M_1, M_2, \ldots, M_n)$ abbreviates $(M_1, (M_2, \ldots, M_n))$. We write our process notation for pattern-matching tuples in the form **tuple** M **as** $(\underline{N}_1, \ldots, \underline{N}_n);P$, where $n > 0$, M is a message (expected to be a tuple), and each $\underline{N}_i$ is an atomic pattern. Let an atomic pattern be either a variable pattern x, or a constant pattern, written $=M$, where M is a message to be

Syntax for Messages and Processes:

a,b,c,k,x,y,z	name
$M,N ::=$	message
x	name: a key or a channel
$\{M\}N$	authenticated encryption of M with key N
(M,N)	message pair
ok	distinguished name
$P,Q,R ::=$	process
out $M(N)$	asynchronous output of N to channel M
in $M(x{:}T);P$	input of x from channel M (x has scope P)
new $x{:}T;P$	fresh generation of name x (x has scope P)
$!P$	unbounded parallel composition of replicas of P
$P \mid Q$	parallel composition of P and Q
0	inactivity
decrypt L **as** $\{y{:}T\}N;P$	bind y to decryption of L with key N (y has scope P)
split M **as** $(x{:}T,y{:}U);P$	solve $(x,y) = M$ (x has scope U and P; y has scope P)
match M **as** $(N,y{:}U);P$	solve $(N,y) = M$ (y has scope P)
C	statement of clause C
expect C	expectation that clause C is derivable

Notations: $(\widetilde{x}{:}\widetilde{T}) \triangleq (x_1{:}T_1,\ldots,x_n{:}T_n)$ and **new** $\widetilde{x}{:}\widetilde{T};P \triangleq$ **new** $x_1{:}T_1;\ldots$ **new** $x_n{:}T_n;P$
Let $S = \{C_1,\ldots,C_n\}$. We write $S \mid P$ for $C_1 \mid \ldots \mid C_n \mid P$.

Rules for Reduction: $P \to P'$

$P \to P' \Rightarrow P \mid Q \to P' \mid Q$	(Red Par)
$P \to P' \Rightarrow$ **new** $x{:}T;P \to$ **new** $x{:}T;P'$	(Red Res)
$P \equiv Q, Q \to Q', Q' \equiv P' \Rightarrow P \to P'$	(Red Struct)
out $a(M) \mid$ **in** $a(x{:}T);P \to P\{M/x\}$	(Red Comm)
decrypt $\{M\}k$ **as** $\{y{:}T\}k;P \to P\{M/y\}$	(Red Decrypt)
split (M,N) **as** $(x{:}T,y{:}U);P \to P\{M/x\}\{N/y\}$	(Red Split)
match (M,N) **as** $(M,y{:}U);P \to P\{N/y\}$	(Red Match)

Structural equivalence $P \equiv Q$ is defined as usual, and $P \to_{\equiv}^{*} P'$ is $P \equiv P'$ or $P \to^{*} P'$.

matched. Each variable pattern translates to a **split**, and each constant pattern translates to a **match**. For example, **tuple** (a,b,c) **as** $(x,{=}b,y);P$ translates to the process **split** $(a,(b,c))$ **as** (x,z);**match** z **as** (b,z);**split** (z,z) **as** $(y,z);P$, where z is fresh. We allow pattern-matching in conjunction with input and decryption processes, and omit type annotations. The technical report has the formal details of these notations.

The presence of statements and expectations in a process induces the following safety properties. Informally, to say an expectation **expect** C is *justified* means there are sufficient statements in parallel to derive C. Then a process is safe if every expectation in every reachable process is justified.

Definition 2 (Safety). *A process P is* safe *iff whenever* $P \to_{\equiv}^{*}$ **new** $\tilde{x}{:}\tilde{T};(\mathbf{expect}\ C \mid P')$ *then* $P' \equiv$ **new** $\tilde{y}{:}\tilde{U};(C_1 \mid \ldots \mid C_n \mid P'')$ *and* $\{C_1,\ldots,C_n\} \models C$ *with* $\{\widetilde{y}\} \cap fn(C) = \varnothing$.

(The definition mentions $\tilde{x}$ to allow fresh names in C, while it mentions $\tilde{y}$ to ensure that the clauses $C, C_1, \ldots, C_n$ all use the same names; the scopes of these names are otherwise irrelevant in the logic.)

Given a process P representing the legitimate participants making up a system, we want to show that no opponent process O can induce P into an unsafe state, where some expectation is unjustified. An opponent is any process within our spi calculus, except it is not allowed to include any expectations itself. (The opponent goal is to confuse the legitimate participants about who is doing what.) As a technical convenience, we require every type annotation in an opponent to be a certain type **Un**; type annotations do not affect the operational semantics, so the use of **Un** does not limit opponent behaviour.

Definition 3 (Opponent). *A process O is an* opponent *iff it contains no expectations, and every type annotation is* **Un**.

Definition 4 (Robust Safety). *A process P is* robustly safe *iff $P \mid O$ is safe for all opponents O.*

For example, the process Q given earlier is robustly safe, because the statements in P suffice to infer Report(alice,42,report42), and they persist in any interaction with an opponent. On the other hand, the process R is safe on its own, but is not robustly safe. Consider the opponent **out** c (bogus,**ok**). We have:

$$\text{R} \mid \textbf{out}\ \text{c}\ (\text{bogus},\textbf{ok}) \rightarrow \text{P} \mid \textbf{out}\ \text{c}\ (\text{report42},\textbf{ok}) \mid \textbf{expect}\ \text{Report(alice,42,bogus)}$$

This is unsafe because Report(alice,42,bogus) is not derivable from the statements in process P. We can secure the channel c by using the **new** operator to make it private. The process **new** c; R is robustly safe; no opponent can inject a message on c.

4 A Type System for Verifying Authorization Assertions

We present a new dependent type system for checking implementations of authorization policies. Our starting point for this development was a type and effect system by Gordon and Jeffrey [15] for verifying one-to-many correspondences. Apart from the new support for logical assertions, the current system features two improvements. First, a new rely-guarantee rule for parallel composition allows us to typecheck a safe process such as $L \mid \textbf{expect}\ L$; the analogous parallel composition cannot be typed in the original system. Second, effects are merged into typing environments, leading to a much cleaner presentation, and to the elimination of typing rules for effect subsumption. We begin by defining the syntax and informal semantics of message types.

Syntax for Types:

$T, U ::=$	type
Un	public data
Ch(T)	channel for T messages
Key(T)	secret key for T plaintext
$(x{:}T, U)$	dependent pair (scope of x is U)
Ok(S)	ok to assume the clauses S

T is *generative* (may be freshly created) iff T is either **Un**, **Key**(U), or **Ch**(U).

Notation: $(x_1{:}T_1, \ldots, x_n{:}T_n, T_{n+1}) \triangleq (x_1{:}T_1, \ldots, (x_n{:}T_n, T_{n+1}))$

A message of type **Un** is public data that may flow to or from the opponent; for example, all ciphertexts are of type **Un**. A message of type $\mathbf{Ch}(T)$ is a name used as a secure channel for messages of type T. Similarly, a message of type $\mathbf{Key}(T)$ is a name used as a secret key for encrypting and decrypting plaintexts of type T. A message of type $(x{:}T, U)$ is a pair (M, N) where M is of type T, and N is of type $U\{M/x\}$. Finally, the token **ok** is the unique message of type $\mathbf{Ok}(S)$, proving S may currently be inferred.

For example, the type $\mathbf{Ch}((x{:}\mathbf{Un}, \mathbf{Ok}(\text{Report}(\text{alice}, 42, x))))$ can be assigned to c in process R, stating that c is a channel for communicating pairs $(M, \mathbf{ok})$ where $M : \mathbf{Un}$ and $\mathbf{ok} : \mathbf{Ok}(\text{Report}(\text{alice}, 42, M))$.

Next, we define typing environments—lists of variable bindings and clauses—plus two auxiliary functions. The function $env(-)$ sends a process to an environment that collects its top-level statements, with suitable name bindings for any top-level restrictions. The function $clauses(-)$ sends an environment to the program consisting of all the clauses listed in the environment plus the clauses in top-level $\mathbf{Ok}(-)$ types.

Syntax for Environments, and Functions $env(P)$ and $clauses(E)$:

$E ::= \varnothing \mid E, x{:}T \mid E, C$ Notation: $E(x) = T$ if $E = E', x{:}T, E''$

E is *generative* iff $E = x_1{:}T_1, \ldots, x_n{:}T_n$ and each T_i is generative.

$env(P \mid Q)^{\widetilde{x},\widetilde{y}} = env(P)^{\widetilde{x}}, env(Q)^{\widetilde{y}}$ (where $\{\widetilde{x},\widetilde{y}\} \cap fn(P \mid Q) = \varnothing$)

$env(\mathbf{new}\ x{:}T; P)^{x,\widetilde{x}} = x{:}T, env(P)^{\widetilde{x}}$ (where $\{\widetilde{x}\} \cap fn(P) = \varnothing$)

$env(!P)^{\widetilde{x}} = env(P)^{\widetilde{x}} \qquad env(C)^{\varnothing} = C \qquad env(P)^{\varnothing} = \varnothing$ otherwise

Convention: $env(P) \triangleq env(P)^{\widetilde{x}}$ for some distinct $\widetilde{x}$ such that $env(P)^{\widetilde{x}}$ is defined.

$clauses(E, C) = clauses(E) \cup \{C\} \quad clauses(E, x{:}\mathbf{Ok}(S)) = clauses(E) \cup S$

$clauses(E, x{:}T) = clauses(E)$ if $T \neq \mathbf{Ok}(S) \quad clauses(\varnothing) = \varnothing$

Our system consists of three judgments, defined by the following tables. The judgments define well-formed environments, types of messages, and well-formed processes.

Rules for Environments and Messages: $E \vdash \diamond$, $E \vdash M : T$

(Env $\varnothing$)
$$\frac{}{\varnothing \vdash \diamond}$$

(Env x)
$$\frac{E \vdash \diamond \quad fn(T) \subseteq dom(E) \quad x \notin dom(E)}{E, x{:}T \vdash \diamond}$$

(Env C)
$$\frac{E \vdash \diamond \quad fn(C) \subseteq dom(E)}{E, C \vdash \diamond}$$

(Msg x)
$$\frac{E \vdash \diamond \quad x \in dom(E)}{E \vdash x : E(x)}$$

(Msg Encrypt)
$$\frac{E \vdash M : T \quad E \vdash N : \mathbf{Key}(T)}{E \vdash \{M\}N : \mathbf{Un}}$$

(Msg Encrypt Un)
$$\frac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un}}{E \vdash \{M\}N : \mathbf{Un}}$$

(Msg Pair)
$$\frac{E \vdash M : T \quad E \vdash N : U\{M/x\}}{E \vdash (M, N) : (x{:}T, U)}$$

(Msg Pair Un)
$$\frac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un}}{E \vdash (M, N) : \mathbf{Un}}$$

(Msg Ok)
$$\frac{E \vdash \diamond \quad fn(S) \subseteq dom(E) \quad clauses(E) \models C \quad \forall C \in S}{E \vdash \mathbf{ok} : \mathbf{Ok}(S)}$$

(Msg Ok Un)
$$\frac{E \vdash \diamond}{E \vdash \mathbf{ok} : \mathbf{Un}}$$

The rule (Msg Ok) populates an $\mathbf{Ok}(S)$ type only if we can infer each clause in the Datalog program S from the clauses in E. For example, if

$$E = \text{alice}{:}\mathbf{Un}, 42{:}\mathbf{Un}, \text{report42}{:}\mathbf{Un}, \text{Referee(alice,42)}, \text{Opinion(alice,42,report42)}$$

then $E \vdash \mathbf{ok} : \mathbf{Ok}(\mathrm{Report}(\mathrm{alice}, 42, \mathrm{report42}))$. The other message typing rules are fairly standard. As in previous systems [16, 15], we need the rules (Msg Encrypt Un), (Msg Pair Un), and (Msg Ok Un) to assign **Un** to arbitrary messages known to the opponent.

Rules for Processes: $E \vdash P$

(Proc Nil) $\dfrac{E \vdash \diamond}{E \vdash \mathbf{0}}$ (Proc Rep) $\dfrac{E \vdash P}{E \vdash !P}$ (Proc Res) $\dfrac{E, x{:}T \vdash P \quad T \text{ generative}}{E \vdash \mathbf{new}\ x{:}T; P}$ (Proc Expect) $\dfrac{E, C \vdash \diamond \quad \mathit{clauses}(E) \models C}{E \vdash \mathbf{expect}\ C}$

(Proc Par) $\dfrac{E, \mathit{env}(Q) \vdash P \qquad E, \mathit{env}(P) \vdash Q \quad \mathit{fn}(P \mid Q) \subseteq \mathit{dom}(E)}{E \vdash P \mid Q}$ (Proc Fact) $\dfrac{E, C \vdash \diamond}{E \vdash C}$

(Proc Decrypt) $\dfrac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Key}(T) \quad E, y{:}T \vdash P}{E \vdash \mathbf{decrypt}\ M\ \mathbf{as}\ \{y{:}T\}N; P}$ (Proc Input) $\dfrac{E \vdash M : \mathbf{Ch}(T) \qquad E, x{:}T \vdash P}{E \vdash \mathbf{in}\ M(x{:}T); P}$

(Proc Decrypt Un) $\dfrac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un} \quad E, y{:}\mathbf{Un} \vdash P}{E \vdash \mathbf{decrypt}\ M\ \mathbf{as}\ \{y{:}\mathbf{Un}\}N; P}$ (Proc Input Un) $\dfrac{E \vdash M : \mathbf{Un} \qquad E, x{:}\mathbf{Un} \vdash P}{E \vdash \mathbf{in}\ M(x{:}\mathbf{Un}); P}$

(Proc Match) $\dfrac{E \vdash M : (x{:}T, U) \quad E \vdash N : T \quad E, y{:}U\{N/x\} \vdash P}{E \vdash \mathbf{match}\ M\ \mathbf{as}\ (N, y{:}U\{N/x\}); P}$ (Proc Output) $\dfrac{E \vdash M : \mathbf{Ch}(T) \quad E \vdash N : T}{E \vdash \mathbf{out}\ M(N)}$

(Proc Match Un) $\dfrac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un} \quad E, y{:}\mathbf{Un} \vdash P}{E \vdash \mathbf{match}\ M\ \mathbf{as}\ (N, y{:}\mathbf{Un}); P}$ (Proc Output Un) $\dfrac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un}}{E \vdash \mathbf{out}\ M(N)}$

(Proc Split) $\dfrac{E \vdash M : (x{:}T, U) \quad E, x{:}T, y{:}U \vdash P}{E \vdash \mathbf{split}\ M\ \mathbf{as}\ (x{:}T, y{:}U); P}$ (Proc Split Un) $\dfrac{E \vdash M : \mathbf{Un} \quad E, x{:}\mathbf{Un}, y{:}\mathbf{Un} \vdash P}{E \vdash \mathbf{split}\ M\ \mathbf{as}\ (x{:}\mathbf{Un}, y{:}\mathbf{Un}); P}$

There are three rules of particular interest. (Proc Expect) allows **expect** C provided C is entailed in the current environment. (Proc Fact) allows any statement, provided its names are in scope. (Proc Par) is a rely-guarantee rule for parallel composition; it allows $P \mid Q$, provided that P and Q are well-typed given the top-level statements of Q and P, respectively. For example, by (Proc Par), $\varnothing \vdash \mathrm{Foo}() \mid \mathbf{expect}\ \mathrm{Foo}()$ follows from $\varnothing \vdash \mathrm{Foo}()$ and $\mathrm{Foo}() \vdash \mathbf{expect}\ \mathrm{Foo}()$, the two of which follow directly by (Proc Fact) and (Proc Expect).

Main Results. Our first theorem is that well-typed processes are safe; to prove it, we rely on a lemma that both structural congruence and reduction preserve the process typing judgment.

Lemma 1 (Type Preservation). *If $E \vdash P$ and either $P \equiv P'$ or $P \to P'$ then $E \vdash P'$.*

Theorem 1 (Safety). *If $E \vdash P$ and E is generative, then P is safe.*

Our second theorem is that well-typed processes whose free names are public, that is, of type **Un**, are robustly safe. It follows from the first via an auxiliary lemma that any opponent process can be typed by assuming its free names are of type **Un**.

Lemma 2 (Opponent Typability). *If* $fn(O) \subseteq \{\widetilde{x}\}$ *for opponent* O *then* $\widetilde{x}{:}\widetilde{\mathbf{Un}} \vdash O$.

Theorem 2 (Robust Safety). *If* $\widetilde{x}{:}\widetilde{\mathbf{Un}} \vdash P$ *then* P *is robustly safe.*

We conclude this section by showing our calculus can encode standard one-to-many correspondence assertions. The idea of correspondences is that processes are annotated with two kinds of labelled events: begin-events and end-events. The intent is that in each run, for every end-event, there is a preceding begin-event with the same label.

We can encode one particular syntax [15] as follows:

$$\textbf{begin}\ !L; P \triangleq L \mid P \qquad\qquad \textbf{end}\ L; P \triangleq \textbf{expect}\ L \mid P$$

With this encoding and a minor extension to the type system (tagged union types), we can express and typecheck all of the authentication protocols from Gordon and Jeffrey's paper [15], including WMF and BAN Kerberos.

The correspondences expressible by standard begin- and end-assertions are a special case of the class of correspondences expressible in our calculus where the predicates in expectations are *extensional*, that is, given explicitly by facts. Hence, we refer to our generalized correspondence assertions based on intensional predicates as *intensional correspondences*, to differentiate them from standard (extensional) correspondences.

5 Application: Access Control for a Programme Committee

We provide two implementations for the Datalog policy with delegation introduced in Section 2 (defining clauses A, B, and C). In both implementations, the server enables those three clauses, and also maintains a local database of registered reviewers, on a private channel pwdb:

```
A | B | C | new pwdb : Ch( u:Un, Key(v:Un,id:Un,Ok(Delegate(u,v,id))),
                                 Key(id:Un,report:Un,Ok(Opinion(u,id,report))));
```

Hence, each message on pwdb codes an entry in the reviewer database, and associates the name u of a reviewer with two keys used to authenticate her two potential actions: delegating a review, and filing a report. The usage of these keys is detailed below.

Although we present our code in several fragments, these fragments should be read as parts of a single process, whose typability and safety properties are summarized at the end of the section. Hence, for instance, our policy and the local channel pwdb are defined for all processes displayed in this section.

Online Delegation, with Local State. Our first implementation assumes the conference system is contacted whenever a referee decides to delegate her task. Hence, the system keeps track of expected reports using a local database, each record showing a fact of the form Referee(U,ID). When a report is received, the authenticated sender of the report is correlated with her record. When a delegation request is received, the corresponding record is updated.

The following code defines the (abstract) behaviour of reviewer v; it is triggered whenever a message is sent on createReviewer; it has public channels providing controlled access to all her privileged actions—essentially any action authenticated with one

of her two keys. For simplicity, we proceed without checking the legitimacy of requests, and we assume v is not a PC member—otherwise, we would implement a third action for filing PC member reports.

```
(!in createReviewer(v);
  new kdv: Key(z:Un,id:Un,Ok(Delegate(v,z,id)));
  new krv: Key(id:Un,report:Un,Ok(Opinion(v,id,report)));
  ( (!out pwdb(v,kdv,krv))
  | (!in sendreportonline(=v,id,report);
       Opinion(v,id,report) | out filereport(v,{id,report,ok}krv) )
  | (!in delegateonline(=v,w,id);
       Delegate(v,w,id) | out filedelegate(v,w,id,{w,id,ok}kdv) ))) |
```

Two new keys are first generated. The replicated output on pwdb associates those keys with v. The replicated input on sendreportonline guards a process that files v's reports; in this process, the encryption {id,report,**ok**}krv protects the report and also carries the fact Opinion(v,id,report) stating its authenticity. The replicated input on delegateonline similarly guards a process that files v's delegations.

Next, we give the corresponding code that receives these two kinds of requests at the server. (We omit the code that selects reviewers and sends message on refereedb.) In the process guarded by !**in** filereport(v,e), the decryption "proves" Opinion(v,id,report), whereas the input on refereedb "proves" Referee(v,id): when both operations succeed, these facts and clause A jointly guarantee that Report(v,id,report) is derivable. Conversely, our type system would catch errors such as forgetting to correlate the paper or the reviewer name (e.g., writing =v,id instead of =v,=id in refereedb), leaking the decryption key, or using the wrong key.

The process guarded by !**in** filedelegate(v,w,id,sigd) is similar, except that it uses the fact Delegate(v,w,id) granted by decrypting under key kdv to transform Referee(v,id) into Referee(w,id), which is expected for typing **ok** in the output on refereedb.

```
new refereedb : Ch( (u:Un,(id:Un,Ok(Referee(u,id)))));
(!in filereport(v,e);
     in pwdb(=v,kdv,krv); decrypt e as {id,report,_}krv;
     in refereedb(=v,=id,_); expect Report(v,id,report)) |
(!in filedelegate(v,w,id,sigd);
     in pwdb(=v,kdv,krv); decrypt sigd as {=w,=id,_}kdv;
     in refereedb(=v,=id,_); out refereedb(w,id,ok)) |
```

Reviews from PC members, using Capabilities. The code for processing PC member reports is similar but simpler:

```
new kp:Key(u:Un,Ok(PCMember(u)));
(!in createPCmember(u,pc);PCMember(u) | out pc({(u,ok)}kp) ) |
(!in filepcreport(v,e,pctoken);
  in pwdb(=v,kdv,krv); decrypt e as {id,report,_}krv;
  decrypt pctoken as {=v,_}kp; expect Report(v,id,report) ) |
```

Instead of maintaining a database of PC members, we (arbitrarily) use capabilities, consisting of the name of the PC member encrypted under a new private key kp. The code also implement two services as replicated inputs, to register a new PC member

and to process a PC member report. The fact Opinion(v,id,report) is obtained as above. Although the capability sent back on channel pc has type **Un**, its successful decryption yields the fact PCMember(v) and thus enables Report(v,id,report) by clause B.

Offline Delegation, with Certificate Chains. The second implementation relies instead on explicit chains of delegation certificates. It does not require that the conference system be contacted when delegation occurs; on the other hand, the system may have to check a list of certificates before accepting an incoming report.

To this end, the process guarded by the replicated input on channel filedelegatereport allocates a private channel link and uses that channel recursively to verify each piece of evidence filed with the report, one certificate at a time. The process guarded by link has two cases: the base case (**decrypt** cu) verifies an initial refereeing request and finally accepts the report as valid; the recursive case (**tuple** cu) verifies a delegation step then continues on the rest of the chain (ct). Note that the type assigned to link precisely states our loop invariant: Delegate(u,v,id) proves that there is a valid delegation chain from u (the report writer) up to v (the current delegator) for paper id.

A further, less important difference is that our second implementation relies on self-authenticated capabilities under key ka for representing initial refereeing requests, instead of messages on the private database channel refereedb. Finally, our second implementation relies on auxiliary clauses making Delegate reflexive and transitive; these clauses give us more freedom but they do not affect the outcome of our policy—one can check that these two clauses are redundant in any derivation of Report.

```
( Delegate(U,W,ID):−Delegate(U,V,ID),Delegate(V,W,ID) ) |
( Delegate(U,U,ID):−Opinion(U,ID,R) ) |
new ka:Key((u:Un,(id:Un,Ok(Referee(u,id))))); 
(!in filedelegatereport(v,e,cv);
  in pwdb(=v,kdv,krv); decrypt e as {id,report,_}krv;
  new link:Ch(u:Un,c:Un,Ok(Delegate(u,v,id))); out link(v,cv,ok) |
  !in link(u,cu,_);
   ( decrypt cu as {=u,=id,_}ka; expect Report(v,id,report)) |
   ( tuple cu as (t,skt,ct);
     in pwdb(=t,kdt,_); decrypt skt as {=u,=id,_}kdt; out link(t,ct,ok)) |
```

Proposition 1. *Let E_P assign the types displayed above to pwdb, refereedb, kp, and ka. Let E_{Un} assign type **Un** to createReviewer, createPCMember, sendreportonline, delegateonline, filereport, filedelegate, filepcreport, filedelegatereport, and any other variable in its domain.*

Let P be a process such that $E_{Un}, E_P \vdash P$. Let Q be the process consisting of all process fragments in this section followed by P.

We have $E_{Un} \vdash Q$, and hence Q is robustly safe.

This proposition is proved by typing Q then applying Theorem 2. In its statement, the process P has access to the private keys and channels collected in E_P; this process accounts for any trusted parts of the server left undefined, including for instance code that assigns papers to reviewers by issuing facts on Referee and using them to populate refereedb and generate valid certificates under key *ka*. We may simply take $P = \mathbf{0}$, or let P introduce its own policy extensions, as long as it complies with the typing environments E_{Un} and E_P.

In addition, the context (implicitly) enclosing Q in our statement of robust safety accounts for any untrusted part of the system, notably the attacker, but also additional code for the reviewers interacting with Q (and possibly P) using the names collected in E_{Un}, and notably the free channels of Q. Hence, the context may impersonate referees, intercept messages on free channels, then send on channel filedelegatereport any term computed from intercepted messages. The proposition confirms that minimal typing assumptions on P suffice to guarantee the robust safety of Q.

6 Application: A Default Implementation for Datalog

We finally describe a translation from Datalog programs to the spi calculus. To each predicate p and arity n, we associate a fresh name p_n with type $T_{p,n}$. Unless the predicate p occurs with different arities, we omit indices and write p and T_p for p_n and $T_{p,n}$. Relying on some preliminary renaming, we also reserve a set of names $\mathscr{V}$ for Datalog variables. The translation is given below:

Translation from Datalog to the spi calculus: $[\![S]\!]$

$T_{p,n} = \mathbf{Ch}(x_1{:}\mathbf{Un},\ldots,x_n{:}\mathbf{Un},\mathbf{Ok}(p(x_1,\ldots,x_n)))$
$[\![S]\!] = \prod_{C\in S}[\![C]\!] \qquad [\![\varnothing]\!] = \mathbf{0} \qquad [\![L{:-}L_1,\ldots,L_m]\!] = ![\![L_1,\ldots,L_m]\!]^{\varnothing}[[\![L]\!]^{+}] \quad \text{for } m \geq 0$
$[\![p(u_1,\ldots,u_n)]\!]^{+} = \mathbf{out}\ p_n(u_1,\ldots,u_n,\mathbf{ok})$
$[\![L_1,L_2,\ldots,L_m]\!]^{\Sigma}[\cdot] = [\![L_1]\!]^{\Sigma}\left[[\![L_2,\ldots,L_m]\!]^{\Sigma\cup fv(L_1)}[\cdot]\right] \qquad [\![\varepsilon]\!]^{\Sigma}[\cdot] = [\cdot]$
$[\![p(u_1,\ldots,u_n)]\!]^{\Sigma}[\cdot] = \mathbf{in}\ p_n(\underline{u}_1,\ldots,\underline{u}_n,{=}\mathbf{ok});[\cdot]$
where $\underline{u_i}$ is u_i when $u_i \notin (\mathscr{V}\setminus(\Sigma\cup fv(u_{j<i})))$ and $\underline{u_i}$ is ${=}u_i$ otherwise.
$P \Downarrow_L$ when $\exists P'.P \rightarrow_{\equiv}^{*} P' \mid [\![L]\!]^{+}$

For example, using the policy of Section 2, the translation of predicate Report uses a channel Report of type $T_{Report} = \mathbf{Ch}(U{:}\mathbf{Un},ID{:}\mathbf{Un},R{:}\mathbf{Un},\mathbf{Ok}(\text{Report(U,ID,R)}))$ and the translation of clause A yields the process

$$[\![\text{Report(U,ID,R)}{:-}\text{Referee(U,ID),Opinion(U,ID,R)}]\!] \quad =$$
$$!\mathbf{in}\ \text{Referee(U,ID,}{=}\mathbf{ok});\ \mathbf{in}\ \text{Opinion(}{=}\text{U,}{=}\text{ID,R,}{=}\mathbf{ok});\ \mathbf{out}\ \text{Report(U,ID,R,}\mathbf{ok})$$

The next lemma states that the translation of a Datalog program is well typed when placed in parallel with itself as a policy.

Lemma 3 (Typability of the encoding). *Let S be a Datalog program using predicates $\widetilde{p_n}$ and names $\widetilde{y}$ with $fn(S) \subseteq \{\widetilde{y}\}$. Let $E = \widetilde{y{:}\mathbf{Un}}, \widetilde{p_n{:}T_{n,p}}$. We have $E \vdash S \mid [\![S]\!]$.*

More precisely, the lemma also shows that our translation is compositional: one can translate some part of a logical policy, develop some specific protocols that comply with some other part of the policy, then put the two implementations in parallel and rely on messages on channels p_n to safely exchange facts concerning shared predicates.

Lemma 3 establishes that our translation is correct by typing. The following theorem also states that the translation is complete: any fact that logically follows from the Datalog program can be observed in the pi calculus.

Theorem 3 (Correctness and completeness). *Let S be a Datalog program and F a fact. We have $S \models F$ if and only if $[\![S]\!] \Downarrow_F$.*

Example. To illustrate our translation, we sketch an alternative implementation of our conference management server. Instead of coding the recursive processing of messages sent by subreferees, as in Section 5, we set up a replicated input for each kind of certificate, with code to check the certificate and send a message on a channel of the translation. Independently, when a fact is expected, we simply read it on a channel of the translation. For instance, to process incoming reports, we may use the code

```
!in trivial_filereport(v,id,report);
 in Report(=v,=id,=report,=ok); expect Report(v,id,report)
```

The translation of clause A sends a matching message on Report, provided the system sends matching messages on Opinion and Referee. This approach is correct and complete, but also non-deterministic and very inefficient. As a refinement, since any (well-typed) program can access the channels of the translation, one may use the translation as a default implementation for some clauses and provide optimized code for others.

7 Conclusions and Future Work

We presented a spi calculus with embedded authorization policies, a type system that can statically check conformance to a policy (even in the presence of active attackers), and a series of applications coded using a prototype implementation.

In itself, our type system does not "solve" authorization: the security of a well-typed program still relies on a careful (manual) review of the policy, on the discriminating statement of trusted facts (or even rules) in the program, and on the presence of **expect** processes marking sensitive actions—indeed, in our setting, every program is safe for a sufficiently permissive policy. Nonetheless, our type system statically enforces a discipline prescribed by the policy across the program, as it uses channels and cryptographic primitives to process messages, and can facilitate code reviews.

Future Work. From a logical viewpoint, many authorization languages extend Datalog with notions of locality and partial trust, considering for examples facts and clauses relative to each principal. Similarly, many variants of the pi calculus feature explicit localities and principals and could, in principle, provide a more realistic distributed semantics for these logics. We are also exploring extensions of our type system to support, for instance, some subtyping, public-key cryptographic primitives, and linearity properties. More experimentally, we plan to extend our typechecker and symbolic interpreter, and to study their integration with other proof techniques.

Acknowledgments. Karthikeyan Bhargavan contributed to several discussions at the start of this project, and commented on a draft of this paper. Martín Abadi and the anonymous conference reviewers made useful suggestions.

References

1. M. Abadi. On SDSI's linked local name spaces. *J. Computer Security*, 6(1–2):3–21, 1998.
2. M. Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, Sept. 1999.
3. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999.
4. M. Y. Becker and P. Sewell. Cassandra: flexible trust management, applied to electronic health records. In *17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pages 139–154, June 2004.
5. B. Blanchet. From secrecy to authenticity in security protocols. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *LNCS*, pages 342–359. Springer, 2002.
6. M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE 17th Symposium on Research in Security and Privacy*, pages 164–173, 1996.
7. C. Braghin, D. Gorla, and V. Sassone. A distributed calculus for role-based access control. In *17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pages 48–60, June 2004.
8. M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: the calculus of boxed ambients. *ACM TOPLAS*, 26(1):57–124, Jan. 2004.
9. M. Bugliesi, D. Colazzo, and S. Crafa. Type based discretionary access control. In *CONCUR'04 - Concurrency Theory*, volume 3170 of *LNCS*, pages 225–239. Springer, Sept. 2004.
10. S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.
11. ContentGuard. XrML 2.0 Technical Overview. `http://www.xrml.org/`, Mar. 2002.
12. J. DeTreville. Binder, a logic-based security language. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 105–113, 2002.
13. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.
14. C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. Technical Report MSR–TR–2005–01, Microsoft Research, 2005.
15. A. D. Gordon and A. Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In *Software Security—Theories and Systems*, volume 2609 of *LNCS*, pages 270–282. Springer, 2002.
16. A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *J. Computer Security*, 11(4):451–521, 2003.
17. A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Comput. Sci.*, 300:379–409, 2003.
18. D. P. Guelev, M. D. Ryan, and P.-Y. Schobbens. Model-checking access control policies. In *Seventh Information Security Conference (ISC'04)*, volume 3225 of *LNCS*. Springer, 2004.
19. J. D. Guttman, F. J. Thayer, J. A. Carlson, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen. Trust management in strand spaces: a rely-guarantee method. In *13th European Symposium on Programming (ESOP'04)*, volume 2986 of *LNCS*, pages 340–354. Springer, 2004.
20. T. Jim. SD3: a trust management system with certified evaluation. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 106–115, 2001.
21. N. Li and J. C. Mitchell. Understanding SPKI/SDSI using first-order logic. In *Proceedings of the 16th IEEE Computer Security Foundation Workshop (CSFW'03)*, pages 89–103, 2003.
22. A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.
23. R. D. Nicola, G. Ferrari, and R. Pugliese. Programming access control: The KLAIM experience. In *CONCUR 2000*, volume 1877 of *LNCS*, pages 48–65. Springer, 2000.
24. T. Woo and S. Lam. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, 1993.

Computationally Sound, Automated Proofs for Security Protocols

Véronique Cortier[1,⋆] and Bogdan Warinschi[2,⋆⋆]

[1] Loria, CNRS, Nancy, France
fax: (+33) 3 83 27 83 19
cortier@loria.fr
[2] Computer Science Department,
University of California at Santa Cruz, USA
bogdan@soe.ucsc.edu

Abstract. Since the 1980s, two approaches have been developed for analyzing security protocols. One of the approaches relies on a computational model that considers issues of complexity and probability. This approach captures a strong notion of security, guaranteed against all probabilistic polynomial-time attacks. The other approach relies on a symbolic model of protocol executions in which cryptographic primitives are treated as black boxes. Since the seminal work of Dolev and Yao, it has been realized that this latter approach enables significantly simpler and often automated proofs. However, the guarantees that it offers have been quite unclear.

In this paper, we show that it is possible to obtain the best of both worlds: fully automated proofs and strong, clear security guarantees. Specifically, for the case of protocols that use signatures and asymmetric encryption, we establish that symbolic integrity and secrecy proofs are sound with respect to the computational model. The main new challenges concern secrecy properties for which we obtain the first soundness result for the case of active adversaries. Our proofs are carried out using Casrul, a fully automated tool.

1 Introduction

Security protocols are short programs designed to achieve various security goals, such as data privacy and data authenticity, even when the communication between parties takes place over channels controlled by an attacker. Their ubiquitous presence in many important applications makes designing and establishing the security of such protocols a very important research goal. Unfortunately, attaining this goal seems to be quite a difficult task, and many of the protocols that had been proposed have been found to be flawed.

Starting in the early '80s, two distinct and quite different methods have emerged in an attempt to ground the security of protocols on firm, rigorous mathematical foundations.

⋆ Véronique Cortier's work was partly supported by the ACI Jeunes Chercheurs Crypto and the RNTL project PROUVE-03V360.

⋆⋆ Bogdan Warinschi was partly supported by the National Science Foudation Grants CCR-0204162 and CCR-0208800.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 157–171, 2005.

They are generically known as the computational (or the cryptographic) approach and the symbolic (or the Dolev-Yao) approach.

Under the computational approach, the security of protocols is based on the security of the underlying primitives, which in turn is proved assuming the hardness of solving various computational tasks such as factoring or taking discrete logarithms. The main tools used for proofs are *reductions*: to prove a protocol secure one shows that a successful adversary against the protocol can be efficiently transformed into an adversary against some primitive used in its construction. Here, quantification is universal over *all* possible probabilistic polynomial-time (p.p.t.) adversaries and the execution model that is analyzed is specified down to the bit-string level. Proofs in the computational model imply strong guarantees (security holds in the presence of an *arbitrary* probabilistic polynomial-time adversary). At the same time however, security reductions for even moderately-sized protocols become extremely long, difficult, and tedious.

The central characteristics of the symbolic approach are an abstract view of the execution and a significantly limited adversary. More precisely, in this model, the implementation details of the primitives are abstracted away, and the execution is modeled only symbolically. Furthermore, the actions of the adversary are quite constrained. For instance, it is postulated that it can recover the plaintext underlying a ciphertext only if it can derive the appropriate decryption key. The resulting execution models are rather simple and can easily be handled by automated tools. In fact, many security proofs have already been carried out using model checkers [16] and theorem provers [19]. Unfortunately, the high degree of abstraction and the limited adversary raise serious questions regarding the security guarantees offered by such proofs, especially from the perspective of the computational model.

Recently, a significant research effort has been directed at bridging the gap between the two approaches [3, 18, 5, 17]. The idea is to determine condition under which symbolic analysis is sound with respect to standard computational models. This path promises tremendous benefits: protocols can be analyzed and proved secure using the simpler, automated methods specific to the symbolic approach, yet the security guarantees are with respect to the more comprehensive computational model. In this paper we extend and apply the work of Micciancio and Warinschi [17] to demonstrate for the first time that *fully automated* security proofs with clear computational implications are indeed possible.

Specifically, our results are as follows. First, we give a language for specifying protocols. The syntax of our language is close to that of Casrul and allows the use of random nonces, digital signatures and public-key encryption. For protocols specified in this language we give two kinds of executions for protocols. Each of these models considers a powerful *active* adversary that controls and potentially tampers with the communication in an unbounded number of sessions of the protocol executed by honest users. The first model is a computational model in which the adversary is an arbitrary p.p.t. algorithm. The second model is symbolic, and the adversary is a typical Dolev-Yao adversary. One crucial property of the latter model is that it actually coincides with the execution semantics used by an existing automated tool called Casrul. We then link the two models in several ways.

Our first contribution (Theorem 1) is a soundness theorem for proofs of trace properties: if *all* symbolic traces of a protocol satisfy a certain predicate (*i.e.* the protocol

is secure in the symbolic model), then the concrete traces satisfy the same predicate with overwhelming probability against p.p.t. adversaries (*i.e.* the protocol is secure in the computational model). Our result is a proper extension of a similar theorem of [17] to protocols that besides nonces and public-key encryption also use digital signatures.

Our second main result concerns soundness of secrecy proofs. This issue is significantly more challenging since unlike in the case of trace properties, secrecy is formalized in quite different ways in the two models that we consider: inability of deriving the secret in the formal world[1] and indistinguishability of adversary's views in the computational world. Nevertheless, we are able to prove that in the case of nonces, symbolic secrecy implies computational secrecy.

Although our theorems justify formal analysis as used in Casrul [9], we also briefly considered other automatic tools, such as Proverif [7], Casper [16], and Securify [10] and we strongly believe that similar soundness results could be obtained for these tools also. While our choice was mainly determined by our familiarity with Casrul (one of the authors is a close collaborator of the team that develops Casrul) an additional factor was that most of the tools dedicated to an unbounded number of sessions allow only for proofs of secrecy and not for authenticity.

Related work. The rationale behind the need for soundness theorems was outlined by Abadi [1] and the first such result was obtained by Abadi and Rogaway [3]. Quite a few other results followed, and here we recall those that are closest to our work. These include the soundness theorem for secrecy properties given by Abadi and Rogaway for symmetric encryption in the presence of passive adversaries [3]. Another results is that of Laud [14] who shows soundness of confidentiality properties for symmetric encryption in a model with a fixed number of sessions. A soundness result for trace properties was proved by Micciancio and Warinschi [17] for a language that used random nonces and public-key encryption. In this paper we extend their work to also include digital signature and ciphertext forwarding. Soundness of trace properties for an even richer language that includes in addition symmetric encryption and authentication was given by Backes, Pfitzmann, and Waidner [5] and work in progress is aimed at achieving soundness for secrecy of symmetric keys [4]. While it is conceivable that building upon these results at least partial automation of symbolic proofs can be achieved, this work still remains to be carried out.

The rest of the paper is structured as follows. In Section 2 we briefly recall digital signatures and public-key encryption schemes. We present the protocol syntax in Section 3 and the two execution models in Section 4. In Section 5 we define generic security properties and prove our soundness theorems for trace and secrecy properties. Section 6 concludes with a discussion regarding the implications of our results on the proofs done with Casrul.

2 Computational Cryptography

In this paper we will use a generic digital signature scheme $\mathcal{DS} = (\mathsf{K_s}, \mathsf{Sig}, \mathsf{Vf})$ given, as usual, by algorithms for key generation, signing and verifying. Also, we consider an

[1] Secrecy can alternatively be defined using an equivalence based formulation, as in the spi-calculus [2] for example, but in this paper we concentrate on the formulation used in Casrul.

arbitrary public-key encryption scheme $\mathcal{AE} = (\mathsf{K_e}, \mathsf{Enc}, \mathsf{Dec})$ given by algorithms for key generation, encryption and decryption. For a precise specification of their syntax we refer to [11].

Traditionally, security is defined for each individual primitive separately. Since the protocols that we aim to analyze may use both encryption and digital signatures, it is more convenient to define the security of signatures and encryption when used simultaneously, in a multi-user environment. We develop a formal model for security that mixes definitional ideas from [13] (for digital signature schemes) and from [20] and [6] (for asymmetric encryption). Here, we only give an overview of the definition. The precise definition can be found in [11]. We consider an experiment parametrized by a digital signature scheme $\mathcal{DS}$, an asymmetric encryption scheme $\mathcal{AE}$, an adversary $\mathcal{A}$, a bit b and a security parameter η. In this experiment the adversary $\mathcal{A}$ has access to an *oracle* denoted $\mathcal{O}_{\mathcal{DS},\mathcal{AE}}(b, \eta)$. The adversary issues the following requests in any order and any number of times:

- creation of keys: the oracle generates (internally) keys for encryption, decryption, signing, and verifying and returns the public keys (*i.e.* keys for encryption and for verifying) to the adversary.
- signature request: the adversary can request signatures on any message it chooses, under any of the secret signing keys that has been generated. The oracle computes such a signature and returns it to the adversary.
- encryption requests: here the adversary submits a pair of messages (m_0, m_1), specifies an encryption key that has been generated and obtains from the oracle the encryption of m_b under that key.
- decryption requests: the adversary can require to see the decryption of any ciphertext of his choosing, provided that the ciphertext has not been obtained from the encryption oracle.

The goal of the adversary is to produce a valid signature on some message which it did not query to the oracle (*i.e.* break the signature scheme), or determine what is the selection bit b with probability significantly better than $1/2$ (*i.e.* break the encryption).

If for all p.p.t. adversaries either of the above events happens only with negligible probability[2] (in the security parameter), then we say that $\mathcal{DS}$ and $\mathcal{AE}$ are jointly secure. Although this is a new measure of security intended for analyzing security of encryption and that of signing when used simultaneously, it is easy to prove that it is implied by standard requirements on the individual primitives. More precisely, it is easy to show that if the digital signature scheme $\mathcal{DS}$ is existentially unforgeable under chosen-message attack [13] and if $\mathcal{AE}$ is secure in the sense of indistinguishability under chosen-ciphertext attacks (IND-CCA) then $\mathcal{DS}$ and $\mathcal{AE}$ are jointly secure.

3 Protocol Syntax

We consider protocols specified in a language similar to the one of Casrul [21] allowing parties to exchange messages built from identities and randomly generated nonces using

[2] A function is said to be negligible if it grows slower than the inverse of any polynomial.

public key encryption and digital signatures. Consider an algebraic signature Σ with the following sorts. A sort ID for agent identities, sorts SKey, VKey, EKey, DKey containing keys for signing, verifying, encryption, and decryption respectively. The algebraic signature also contains sorts Nonce, Label, Ciphertext, Signature, and Pair for respectively nonces, labels, ciphertexts, signatures, and pair. The sort Label is used in encryption and signatures to distinguish between different encryption/signature of the same plaintext. The sort Term is a supersort containing all other sorts, except SKey and DKey. There are nine operations: the four operations ek, dk, sk, vk are defined on the sort ID and return the encryption key, decryption key, signing key, and verification key associated to the input identity. The two operations ag and adv are defined on natural numbers and return labels: these labels are used to differentiate between different encryptions (and signatures) of the same plaintext, created by the honest agents or the adversary. We distinguish between labels for agents and for the adversary since they do not use the same randomness. The other operations that we consider are pairing, public key encryption, and signing with the following ranges and domains.

- $\langle \_ , \_ \rangle$: Term $\times$ Term $\rightarrow$ Pair
- $\{\_\}^{\_}_{\_}$: EKey $\times$ Term $\times$ Label $\rightarrow$ Ciphertext
- $[\_]^{\_}_{\_}$: SKey $\times$ Term $\times$ Label $\rightarrow$ Signature

Protocols are specified using the algebra of terms constructed over the above signature from a set X of sorted variables. Specifically, $\mathsf{X} = \mathsf{X}.n \cup \mathsf{X}.a \cup \mathsf{X}.c \cup \mathsf{X}.s \cup \mathsf{X}.l$, where $\mathsf{X}.n, \mathsf{X}.a, \mathsf{X}.c, \mathsf{X}.s, \mathsf{X}.l$ are sets of variables of sort nonce, agent, ciphertext, signature, and labels respectively. Furthermore, $\mathsf{X}.a$ and $\mathsf{X}.n$ are as follows. If $k \in \mathbb{N}$ is some fixed constant representing the number of protocol participants, w.l.o.g. we fix the set of agent variables to be $\mathsf{X}.a = \{A_1, A_2, \ldots, A_k\}$, and partition the set of nonce variables, by the party that generates them. Formally: $\mathsf{X}.n = \cup_{A \in \mathsf{X}.a} \mathsf{X}_n(A)$ and $\mathsf{X}_n(A) = \{X_A^j \mid j \in \mathbb{N}\}$. This partition avoids to specify later, for each role, which variables stand for generated nonces and which variables stand for expected nonces.

The messages that are sent by participants are specified using terms in $\mathsf{T}_\Sigma(\mathsf{X})$, the free algebra generated by X over the signature Σ. The individual behavior of each protocol participant is defined by a *role* that describes a sequence of message receptions/transmissions. A k-party protocol is given by k such roles.

Definition 1 (Roles and protocols). *The set* Roles *of roles for protocol participants is defined by* $\mathsf{Roles} = ((\{\mathsf{init}\} \cup \mathsf{T}_\Sigma(\mathsf{X})) \times (\mathsf{T}_\Sigma(\mathsf{X}) \cup \{\mathsf{stop}\}))^*$.
A k-party protocol is a mapping $\Pi : [k] \rightarrow \mathsf{Roles}$, where $[k]$ denotes the set $\{1, 2, \ldots, k\}$.

We assume that a protocol specification is such that $\Pi(j) = ((l_1^j, r_1^j), (l_2^j, r_2^j), \ldots)$, the j'th role in the definition of the protocol being executed by player A_j. Each sequence $((l_1, r_1), (l_2, r_2), \ldots) \in \mathsf{Roles}$ specifies the messages to be sent/received by the party executing the role: at step i, the party expects to receive a message conforming to l_i and returns message r_i. We wish to emphasize however that terms l_i^j, r_i^j are not actual messages but specify how the message that is received and the message that is output should look like.

Example 1. The Needham-Schroeder-Lowe protocol [15] is specified as follows: there are two roles $\Pi(1)$ and $\Pi(2)$ corresponding to the sender's role and the receiver's role.

$$\begin{aligned} A &\to B: \ \{N_a, A\}_{\mathsf{ek}(B)} \\ B &\to A: \ \{N_a, N_b, B\}_{\mathsf{ek}(A)} \\ A &\to B: \ \{N_b\}_{\mathsf{ek}(B)} \end{aligned}$$

$$\Pi(1) = (\mathsf{init}, \{X^1_{A_1}, A_1\}^{\mathsf{ag}(1)}_{\mathsf{ek}(A_2)}), (\{X^1_{A_1}, X^1_{A_2}, A_2\}^{L}_{\mathsf{ek}(A_1)}, \{X^1_{A_2}\}^{\mathsf{ag}(1)}_{\mathsf{ek}(A_2)})$$
$$\Pi(2) = (\{X^1_{A_1}, A_1\}^{L_1}_{\mathsf{ek}(A_2)}, \{X^1_{A_1}, X^1_{A_2}, A_2\}^{\mathsf{ag}(1)}_{\mathsf{ek}(A_1)}), (\{X^1_{A_2}\}^{L_2}_{\mathsf{ek}(A_2)}, \mathsf{stop})$$

EXECUTABLE PROTOCOLS. Clearly, not all protocols written using the syntax above are meaningful. We only consider the class of *executable protocols*, *i.e.* protocols for each role can be implemented in an executable program, using only the local knowledge of the corresponding agent. This requires in particular that any sent message (corresponding to some r^j_k) is always deducible from the previously received messages (corresponding to $l^j_1, \ldots, l^j_k$). A precise definition may found in [11].

4 Execution Models

In this section we give a symbolic and a computational execution model for the protocols specified using the syntax defined in the previous section. In the symbolic model the honest parties and the adversary exchange elements of a certain term algebra; the adversary can compute its messages only following the standard Dolev-Yao restrictions. In the concrete execution model, the messages that are exchanged are bit-strings and the honest parties and the adversary are p.p.t. Turing machines.

4.1 Formal Execution Model

In the formal execution model, messages are terms of the free algebra T^f defined by:

$$\begin{aligned} T^f ::= \mathbb{N} \mid a \mid \mathsf{ek}(a) \mid \mathsf{dk}(a) \mid \mathsf{sk}(a) \mid \mathsf{vk}(a) \mid n(a,j,s) & \qquad a \in \mathsf{ID}, j, s \in \mathbb{N} \\ \langle T^f, T^f \rangle \mid \{T^f\}^{\mathsf{ag}(i)}_{\mathsf{ek}(a)} \mid \{T^f\}^{\mathsf{adv}(i)}_{\mathsf{ek}(a)} \mid [T^f]^{\mathsf{ag}(i)}_{\mathsf{sk}(a)} \mid [T^f]^{\mathsf{adv}(i)}_{\mathsf{sk}(a)} & \qquad a \in \mathsf{ID}, i \in \mathbb{N} \end{aligned}$$

If A is a variable, or constant of sort agent, we define its knowledge by $\mathbf{kn}(A) = \{\mathsf{dk}(A), \mathsf{sk}(A)\} \cup \mathsf{X}_n(A)$ *i.e.* an agent knows its secret decryption and signing key as well as the nonces it generates during the execution. The formal execution model is a state transition system. A *global state* of the system is given by (Sid, f, H) where H is a set of terms of T^f representing the messages sent on the network and f maintains the local states of all sessions ids Sid. Session identities are tuples of the form $(n, j, (a_1, a_2, \ldots, a_k)) \in (\mathbb{N} \times \mathbb{N} \times \mathsf{ID}^k)$, where $n \in \mathbb{N}$ identifies the session, the names $a_1, a_2, \ldots, a_k$ are the identities of the parties that are involved in the protocol and j is the index of the role that is executed in this session. Mathematically, f is a function $f : \mathsf{Sid} \to ([\mathsf{X} \to T^f] \times \mathbb{N} \times \mathbb{N})$, where $f(\mathsf{sid}) = (\sigma, i, p)$ is the local state of session sid. The function σ is a partial instantiation of the variables occurring in role $\Pi(i)$ and $p \in \mathbb{N}$ is the control point of the program. Three transitions are allowed.

$$\frac{}{S \vdash m}\, m \in S \qquad \frac{}{S \vdash b, \mathsf{ek}(b), \mathsf{vk}(b)}\, b \in \mathsf{X}.a \qquad \text{Initial knowledge}$$

$$\frac{S \vdash m_1 \quad S \vdash m_2}{S \vdash \langle m_1\, , m_2 \rangle} \qquad \frac{S \vdash \langle m_1\, , m_2 \rangle}{S \vdash m_i}\, i \in \{1,2\} \qquad \text{Pairing and unpairing}$$

$$\frac{S \vdash \mathsf{ek}(b) \quad S \vdash m}{S \vdash \{m\}^{\mathsf{adv}(i)}_{\mathsf{ek}(b)}}\, i \in \mathbb{N} \qquad \frac{S \vdash \{m\}^{l}_{\mathsf{ek}(b)} \quad S \vdash \mathsf{dk}(b)}{S \vdash m} \qquad \text{Encryption and decryption}$$

$$\frac{S \vdash \mathsf{sk}(b) \quad S \vdash m}{S \vdash [m]^{\mathsf{adv}(i)}_{\mathsf{sk}(b)}}\, i \in \mathbb{N} \qquad \frac{S \vdash [m]^{\mathsf{ag}(i)}_{\mathsf{sk}(b)}}{S \vdash [m]^{\mathsf{adv}(j)}_{\mathsf{sk}(b)}}\, i,j \in \mathbb{N} \qquad \frac{S \vdash [m]^{l}_{\mathsf{sk}(b)}}{S \vdash m} \qquad \text{Signature}$$

Fig. 1. Deduction rules for the formal adversary; here S is an arbitrary set of formal terms

- $(\mathsf{SId}, f, H) \xrightarrow{\mathbf{corrupt}(a_1,\ldots,a_l)} (\mathsf{SId}, f, \cup_{1\le j\le l}\mathbf{kn}(a_j) \cup H)$. The adversary corrupts parties by outputting a set of identities. He receives in return the secret keys corresponding to the identities. It happens only once at the beginning of the execution.
- The adversary can initiate new sessions: $(\mathsf{SId}, f, H) \xrightarrow{\mathbf{new}(i,a_1,\ldots,a_k)} (\mathsf{SId}', f', H')$ where H', f' and SId' are defined as follows. Let $s = |\mathsf{SId}| + 1$, be the session identifier of the new session, where $|\mathsf{SId}|$ denotes the cardinality of SId. H' is defined by $H' = H \cup \{(s, i, (a_1, \ldots, a_k))\}$ and $\mathsf{SId}' = \mathsf{SId} \cup \{(s, i, (a_1, \ldots, a_k))\}$. The function f' is defined as follows.
  - $f'(\mathsf{sid}) = f(\mathsf{sid})$ for every $\mathsf{sid} \in \mathsf{SId}$.
  - $f'(s, i, (a_1, \ldots, a_k)) = (\sigma, i, 1)$ where σ is a partial function $\sigma : \mathsf{X} \to T^f$ and:

$$\begin{cases} \sigma(A_j) = a_j & 1 \le j \le k \\ \sigma(X^j_{A_i}) = n(a_i, j, s) & j \in \mathbb{N} \end{cases}$$

  We recall that the principal executing the role $\Pi(i)$ is represented by A_i thus, in that role, every variable of the form $X^j_{A_i}$ represents a nonce generated by A_i.
- The adversary can send messages: $(\mathsf{SId}, f, H) \xrightarrow{\mathbf{send}(\mathsf{sid},m)} (\mathsf{SId}, f', H')$ where $\mathsf{sid} \in \mathsf{SId}$, $m \in T^f$, H', and f' are defined as follows. We define $f'(\mathsf{sid}') = f(\mathsf{sid}')$ for every $\mathsf{sid}' \neq \mathsf{sid}$. We denote $\Pi(j) = ((l^j_1, r^j_1), \ldots, (l^j_{k_j}, r^j_{k_j}))$. $f(\mathsf{sid}) = (\sigma, j, p)$ for some σ, j, p. There are two cases.
  - Either there exists a least general unifier θ of m and $l^j_p\sigma$. Then $f'(\mathsf{sid}) = (\sigma \cup \theta, i, p+1)$ and $H' = H \cup \{r^j_p\sigma\theta\}$.
  - Or we define $f'(\mathsf{sid}) = f(\mathsf{sid})$ and $H' = H$ (the state remains unchanged).

If we denote by $\mathsf{SID} = \mathbb{N} \times \mathbb{N} \times \mathsf{ID}^k$ the set of all sessions ids, the set of *symbolic execution traces* is $\mathsf{SymbTr} = \mathsf{SID} \times (\mathsf{SID} \to ([\mathsf{X} \to T^f] \times \mathbb{N} \times \mathbb{N})) \times 2^{T^f}$.

The adversary intercepts messages between honest participants and computes new messages using the deduction relation $\vdash$ defined in Figure 1. Intuitively, $S \vdash m$ means that the adversary is able to compute the message m from the set of messages S. All

deduction rules are rather standard with the exception of the last two; for these rules some explanations are in order. The next to last rule states that given a signature on some message m, the adversary can compute new signatures on the same message. The last rules states that the adversary can recover the corresponding message out of a given signature. Both rules are needed to obtain soundness. The rules reflect capabilities that do not contradict the standard computational security definition of digital signatures, and thus are available to computational adversaries.

Then, a symbolic execution trace $(\mathsf{Sid}_1, f_1, H_1), \ldots, (\mathsf{Sid}_n, f_n, H_n)$ is *valid* if the messages sent by the adversary can be computed by Dolev-Yao operations, *i.e.* if, whenever $(\mathsf{Sid}_i, f_i, H_i) \xrightarrow{\mathbf{send}(s,m)} (\mathsf{Sid}_{i+1}, f_{i+1}, H_{i+1})$, we have $H_i \vdash m$. Given a protocol Π, the set of valid symbolic execution traces is denoted by $\mathsf{Exec}^s(\Pi)$.

Example 2. Playing with the Needham-Schroeder-Lowe protocol described in Example 1, an adversary can corrupt an agent a_3, start a new session for the second role with players a_1, a_2 and send the message $\{n(a_3,1,1), a_1\}^{\mathsf{adv}(1)}_{\mathsf{ek}(a_2)}$ to the player of the second role. The corresponding valid trace execution is:

$$(\emptyset, f_1, \emptyset) \xrightarrow{\mathbf{corrupt}(a_3)} (\emptyset, f_1, \mathbf{kn}(a_3)) \xrightarrow{\mathbf{new}(2,a_1,a_2)} (\{\mathsf{sid}_1\}, f_2, \mathbf{kn}(a_3) \cup \{\mathsf{sid}_1\})$$
$$\xrightarrow{\mathbf{send}(\mathsf{sid}_1, \{n_3,a_1\}^{\mathsf{adv}(1)}_{\mathsf{ek}(a_2)})} \left(\{\mathsf{sid}_1\}, f_3, \mathbf{kn}(a_3) \cup \{\mathsf{sid}_1, \{n_3, n_2, a_2\}^{\mathsf{ag}(1)}_{\mathsf{ek}(a_1)}\}\right),$$

where $\mathsf{sid}_1 = (1, 2, (a_1, a_2))$, $n_2 = n(a_2, 1, 1)$, $n_3 = n(a_3, 1, 1)$, and f_2, f_3 are defined as follows: $f_2(\mathsf{sid}_1) = (\sigma_1, 2, 1)$, $f_3(\mathsf{sid}_1) = (\sigma_2, 2, 2)$ where $\sigma_1(A_1) = a_1$, $\sigma_1(A_2) = a_2$, $\sigma_1(X^1_{A_2}) = n_2$, and σ_2 extends σ_1 by $\sigma_2(X^1_{A_1}) = n_3$.

4.2 Concrete Execution Model

In a concrete execution, the messages that are exchanged are bit-strings and depend on a security parameter η (which is used for example to determine the length of random nonces). We denote by $\mathcal{C}^\eta$ the set of valid messages. We denote the subsets containing values for agent identities, nonces, encryption keys, verification keys, ciphertexts, signatures, and pairs by $C^\eta.a, \mathcal{C}^\eta.n, \mathcal{C}^\eta.e, \mathcal{C}^\eta.v, \mathcal{C}^\eta.c, \mathcal{C}^\eta.s, \mathcal{C}^\eta.p$ respectively. The implementation is such that each bit-string in $\mathcal{C}^\eta$ has a unique type which can be efficiently recovered by using the function $\mathsf{type} : \mathcal{C}^\eta \to \{a, n, e, v, c, s, p\}$. The operations are implemented as follows: we assume a PKI-like setting in which the public keys of parties (those for encryption and signature verification) are accessible to all parties. We model this situation by making available to all parties the (efficiently invertible and) publicly computable functions $\mathsf{vk} : \mathcal{C}^\eta.a \to \mathcal{C}^\eta.v$ and $\mathsf{ek} : \mathcal{C}^\eta.a \to \mathcal{C}^\eta.e$ which given an agent identity return its signature verification key and encryption key respectively. In the concrete implementation, encryption, and signing are implemented with encryption scheme $\mathcal{AE} = (\mathsf{K_e}, \mathsf{Enc}, \mathsf{Dec})$ and digital signature scheme $\mathcal{DS} = (\mathsf{K_s}, \mathsf{Sig}, \mathsf{Vf})$, which we fix throughout this section. Pairing is implemented by some standard (efficiently invertible) encoding function $\langle \cdot\,, \cdot \rangle : \mathcal{C}^\eta \times \mathcal{C}^\eta \to \mathcal{C}^\eta.p$.

The global state of the execution is a pair (f, Sid), where f is used to represent the local state of each session, and Sid represents the set of session ids.

Session ids are tuples $(n, i, (a_1, a_2, \ldots, a_l))$, where $n \in \mathbb{N}$ is a unique session identifier, i is the index of the role executed in this session and $a_1, a_2, \ldots a_k \in \mathcal{C}^\eta$ are the names of the agents involved in running this session. The state function $f : \mathsf{SId} \to [\mathsf{X} \to \mathcal{C}^\eta] \times \mathbb{N} \times \mathbb{N}$, given a session id sid returns $f(\mathsf{sid}) = (\sigma, i, p)$ where σ assigns values to the variables of the program executed in this session (see the discussion regarding the execution of individual roles), i is the index of the role executed in this session and p is the program counter that keeps track of the next step to be executed in this session.

We now discuss how the execution proceeds in this setting.

- At the beginning of the execution, the adversary corrupts a set of parties via a request $\mathbf{corrupt}(a_1, a_2, \ldots)$, where $a_1, a_2, \ldots \in \mathcal{C}^\eta.a$ are agent identities. As a result, the key generation algorithms for encryption and signing are executed, the public keys are published and the secrets keys are given to the adversary.
- The adversary initiates new sessions by issuing requests $\mathbf{new}(i, a_1, \ldots, a_k)$, with $i \in [k]$ and $a_1, \ldots, a_k \in \mathcal{C}^\eta.a$. In this case, cryptographic keys are generated for those agents which do not have such keys, the (public) encryption and verification keys are published and a new session is initiated: if (SId, f) is the state of the execution prior to the request the resulting state is (SId', f') with $\mathsf{SId}' = \mathsf{SId} \cup \{\mathsf{sid}\}$, $\mathsf{sid} = (|\mathsf{SId}| + 1, i, (a_1, \ldots, a_k))$, and f' defined as follows:
  - $f'(s) = s$ for $s \in \mathsf{SId}$ (*i.e.* the local states of previous sessions stay unchanged)
  - $f'(\mathsf{sid}) = (\sigma, i, 1)$ with $\sigma : \mathsf{X} \to \mathcal{C}^\eta$ defined as follows:

$$\begin{cases} \sigma(A_j) & = a_j & 1 \le j \le k \\ \sigma(X^j_{A_i}) & = n(a_i, j, s) \xleftarrow{\$} \mathcal{C}^\eta.n & j \in \mathbb{N} \end{cases}$$

  The local state of the new session is initialized by mapping agent variables to the names of the agents selected by the adversary, and selecting random values for the nonces generated by the party executing the role.
  In addition, for each term $\{t\}^l_{\mathsf{ek}(A_j)}$ and each term $[t]^l_{\mathsf{sk}(A_j)}$ that are sent (*i.e.* occurring within some r^j_i of $\Pi(i)$) we choose random coins $re^{\mathsf{sid}}(t, A_j, l)$ and $rs^{\mathsf{sid}}(t, A_j, l)$ respectively. These coins will later be used in randomizing the encryption and signing functions in the concrete implementation.
- The third kind of queries are message transmission queries $\mathbf{send}(\mathsf{sid}, m)$, with $\mathsf{sid} \in \mathsf{SId}$ and $m \in \mathcal{C}^\eta$ which are processed in two steps:
  First, the incoming message is parsed as an instantiation of the term l^p_i, where we let (σ, i, p) be the local state $f(\mathsf{sid})$ of session sid prior to the request. The parsing is done recursively, on the structure of l^p_i, and the final result is a mapping σ' assigning values in $\mathcal{C}^\eta$ to the variables occurring in l^p_i. To facilitate the parsing procedure, we assume that 1) from any valid ciphertext it is easy to recover the key used for encryption (which is public) and 2) from any valid signature, it is easy to recover the message that was signed and the verification key that needs to be used for verifying. Both these requirements can be easily achieved by tagging the signatures and the ciphertext with the appropriate information.

  In the second step, the local state of sid is updated and a protocol message is computed and returned to the adversary. If the parsing procedure fails at any point (the types of the term and of the bit-string do not match, or a ciphertext is invalid

etc) then the local state of sid remains unchanged. This is also the case if there exists some variable $X \in \mathsf{X}$ for which σ and σ' assign different values. Otherwise, the local store is updated to $\sigma = \sigma \cup \sigma'$ and the answer is computed by replacing each variable X in r_i^p with $\sigma(X)$ and replacing the encryptions and signatures with their computational counterparts, *i.e.* with the randomized functions Enc and Sig.

The execution model that we described above uses randomization: the adversary is probabilistic, and the honest parties use randomization for generating nonces, encryptions, and signatures. It can be shown that if the adversary A runs in polynomial-time, then the honest parties use a number of coins that is a polynomial in the security parameter. In the following, for a fixed adversary $\mathcal{A}$ we denote by $\{0,1\}^{p_{\mathcal{A}}(\eta)}$, resp. by $\{0,1\}^{g_{\mathcal{A}}(\eta)}$, the spaces from where the adversary, resp. the honest parties, draw the coins used in the execution. Notice that each pair of random coins $(R_{\mathcal{A}}, R_{\Pi}) \in \{0,1\}^{p_{\mathcal{A}}(\eta)} \times \{0,1\}^{g_{\mathcal{A}}(\eta)}$ determines a unique sequence of global states (f_1, SId_1), $(f_2, \mathsf{SId}_2), \ldots$, called the *concrete trace* determined by random coins $(R_{\Pi}, R_{\mathcal{A}})$ and which we denote by $\mathsf{Exec}_{\Pi(R_{\Pi}),\mathcal{A}(R_{\mathcal{A}})}(\eta)$. If the set of all possible session ids is $\mathcal{SId} = \mathbb{N} \times [k] \times (\mathcal{C}^{\eta}.a)^k$ then, we denote by ConcTr the set of all possible concrete traces: $\cup_{\eta}(\mathcal{SId} \times [\mathcal{SId} \to [\mathsf{X} \to \mathcal{C}^{\eta}])^*$.

5 Security Properties and Soundness Theorems

We are interested in two types of security properties. Integrity properties and secrecy properties. The former are quite general: for example, they encompass various forms of authentication (both for messages and entities). Our focus will be secrecy properties: we give formalizations for this kind of properties in both the formal and in the computational model, focusing on nonces. We then prove our second main result, a soundness theorem for secrecy of nonces.

5.1 Relating Symbolic and Concrete Traces

Concrete traces can be regarded as instantiations of formal traces via appropriate instantiations of the terms. More precisely, given a formal trace $t^s = (\mathsf{Sld}_1^s, f_1, H_1), \ldots, (\mathsf{Sld}_n^s, f_n, H_n)$, one can obtain a concrete execution trace $t^c = (\mathsf{Sld}_1^c, g_1), \ldots, (\mathsf{Sld}_n^c, g_n)$ on the following way. Once an injective function $c : T^f \to \mathcal{C}^{\eta}$ that maps terms to bitstrings is chosen, t^c is obtained by instantiating the local states: if $f_i(\mathsf{sid}) = (\sigma^{\mathsf{sid}}, i^{\mathsf{sid}}, p^{\mathsf{sid}})$ then $g_i(\mathsf{sid}) = (\tau^{\mathsf{sid}}, i^{\mathsf{sid}}, p^{\mathsf{sid}})$ where $\tau^{\mathsf{sid}} = c \circ \sigma^{\mathsf{sid}}$, and the session ids are unchanged: $\mathsf{Sld}_i^s = \mathsf{Sld}_i^c$. In that case, we say that t^c is a *concrete instantiation* of t^s (or alternatively t^s is a *symbolic representation* of t^c) and we write $t^s \preceq t^c$.

For $\mathsf{P} \subseteq \mathsf{SymbTr}$ we denote by $\mathsf{concrete}(\mathsf{P})$ the set $\{t^c \mid \exists t^s \in \mathsf{P}\ t^s \preceq t^c\}$ of all concrete instantiations of symbolic traces in P.

Technically, the following lemma is at the core of our results. It states that with overwhelming probability, the concrete executions traces of a protocol are instantiations of *valid* symbolic execution traces.

Lemma 1. *Let Π be an executable protocol. If in the concrete implementation the schemes $\mathcal{AE}$ and $\mathcal{DS}$ are jointly secure then for any p.p.t. algorithm $\mathcal{A}$*

$$\Pr\left[\exists t^s \in \mathsf{Exec}^s(\Pi) \mid t^s \preceq \mathsf{Exec}^c_{\Pi(R_{\Pi}),\mathcal{A}(R_{\mathcal{A}})}(\eta)\right] \geq 1 - \nu_{\mathcal{A}}(\eta)$$

where the probability is over the choice $(R_\Pi, R_\mathcal{A}) \stackrel{\$}{\leftarrow} \{0,1\}^{p_\mathcal{A}(\eta)} \times \{0,1\}^{g_\mathcal{A}(\eta)}$ *and* $\nu_\mathcal{A}(\cdot)$ *is some negligible function.*

Proof (Overview). Due to space constraints we only sketch the main aspects of the proof (details may be found in [11]).

The proof works in two steps. First, we explain how each concrete execution trace $\mathsf{Exec}^c_{\Pi(R_\Pi),\mathcal{A}(R_\mathcal{A})}$ determines a unique symbolic trace t^s. We construct t^s by tracing the queries made by the concrete adversary $\mathcal{A}$ and translating them into symbolic queries. Specifically, we map each bit-string m occurring in the execution to a symbolic term $\mathsf{c}(m)$ as follows. Agent identities, cryptographic keys and random nonces (which are quantities that are uniquely determined by R_Π) are canonically mapped to symbolic representations: for example the bit-string representing the decryption key of party a_i is mapped to $\mathsf{sk}(a_i)$. The rest of the messages are interpreted as they occur: each message m sent by the adversary is parsed (notice that all keys that are needed are already known) and its symbolic interpretation is obtained by replacing all occurring basic values (keys, nonces, identities) with their symbolic interpretation, and then replacing the concrete operations with their symbolic counterparts.

In the second step of the proof, we show that with overwhelming probability over the choice of $(R_\Pi, R_\mathcal{A})$, the trace t^s obtained as explained above is a valid execution trace. We prove this statement by contradiction: given an adversary $\mathcal{A}$ we construct three adversaries $\mathcal{B}_1, \mathcal{B}_2$ and $\mathcal{B}_3$ such that if with non-negligible probability the symbolic trace associated to the execution of $\mathcal{A}$ is not a valid Dolev-Yao trace, then at least one of the three adversaries breaks the joint security of $\mathcal{DS}$ and $\mathcal{AE}$.

The idea behind the construction of these adversaries is to execute adversary $\mathcal{A}$ as a subroutine, and use access to the oracle $\mathcal{O}_{\mathcal{DS},\mathcal{AE}}$ (to which each of the three adversaries has access) to simulate the execution of the protocol on behalf of the honest parties. Then, we show that, using the invalid query made by $\mathcal{A}$, adversary $\mathcal{B}_i$ (with $i = 1, 2, 3$) can break either the encryption, or the signing scheme, each of the three adversaries exploiting one of the following three possibilities. Adversary $\mathcal{B}_1$ is based on the assumption that the invalid query of adversary $\mathcal{A}$ contains a signature $[t]_{\mathsf{sk}(a_i)}$ under the secret key of an honest party a_i which was never sent prior in the execution. This essentially means that the corresponding concrete term is a signature forgery, and adversary $\mathcal{B}_1$ simply outputs it. Adversaries $\mathcal{B}_2$ and $\mathcal{B}_3$ correspond to the case where the adversary $\mathcal{A}$ outputs the encryption of some term t such that neither t nor the encryption can be computed by the adversary from the previous messages using only Dolev-Yao operations. In this case we show how to use the adversary $\mathcal{A}$ to determine some secret which he should not have been able to compute. This secret is a random nonce generated by some honest party in the case of adversary $\mathcal{B}_2$ and a signature also generated by an honest party, in the case of adversary $\mathcal{B}_3$. Moreover, the adversaries $\mathcal{B}_1, \mathcal{B}_2$, and $\mathcal{B}_3$ that we construct are such that their sample space partition the sample space of the experiment in which adversary $\mathcal{A}$ is executed. Therefore, if with non-negligible probability the adversary $\mathcal{A}$ has an invalid symbolic execution trace, then with non-negligible probability at least one of the adversaries $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$ breaks the joint security of $\mathcal{DS}$ and $\mathcal{AE}$ which contradicts the hypothesis of the theorem. □

5.2 Trace Properties

For both the symbolic and the computational execution model, trace properties are predicates on the global execution traces. The definition of security (i.e. when a protocol satisfies a given trace property) differs between the symbolic and the computational model. We now give these definitions and give our main result: a soundness theorem for proofs of trace properties.

SYMBOLIC TRACE PROPERTIES. A symbolic trace property is a predicate on (or alternatively a subset of) the set SymbTr. We say that protocol Π satisfies the symbolic trace property $\mathsf{P}^s \subseteq \mathsf{SymbTr}$ and we write $\Pi \models^s \mathsf{P}^s$, if all valid execution traces satisfy P^s, *i.e.* $\mathsf{Exec}^s(\Pi) \subseteq \mathsf{P}^s$.

Various definitions of authentication may be expressed using such trace properties. Informally, a trace of a protocol is a "good" mutual entity authentication trace, if for any two identities a and b, if a (playing the second role of the protocol) has finished a session of the protocol with intended partner b (playing the first role of the protocol), then b has finished a session with intended partner a. Using this characterization, we say that a protocol is a secure authentication protocol if all its traces are good. Depending on which notion of authentication we consider, we may also require that for any session where a terminates, there exists exactly one corresponding session where b terminates and b must have finished before a.

COMPUTATIONAL TRACE PROPERTIES. A computational trace property is a predicate on ConcTr. We say that protocol Π satisfies the concrete security property $\mathsf{P}^c \subseteq \mathsf{ConcTr}$, and we write $\Pi \models^c \mathsf{P}^c$ if its execution traces satisfy P^c with overwhelming probability over the coins used in the execution, *i.e.* for every p.p.t. adversary $\mathcal{A}$, the probability $\Pr\left[\mathsf{Exec}_{\Pi(R_\Pi),\mathcal{A}(R_\mathcal{A})}(\eta) \notin \mathsf{P}^c\right]$ is negligible as a function of η. The probability is taken over the choice $(R_\Pi, R_\mathcal{A}) \stackrel{\$}{\leftarrow} \{0,1\}^{p_\mathcal{A}(\eta)} \times \{0,1\}^{q_\mathcal{A}(\eta)}$.

For mutual authentication, good traces are those satisfying the predicate we sketched for the symbolic model, but the definition of security for protocols is specific to the computational setting: it asks from protocol to have good traces with overwhelming probability. It thus allows for "bad" runs, but only with negligible probability.

One of our contributions is the following soundness theorem for trace properties.

Theorem 1. *Let Π be an executable protocol, $\mathsf{P}^s \subseteq \mathsf{SymbTr}$ be an arbitrary symbolic trace property and $\mathsf{P}^c \subseteq \mathsf{ConcTr}$ be a computational security property such that $\mathsf{concrete}(\mathsf{P}^s) \subseteq \mathsf{P}^c$. Then $\Pi \models^s \mathsf{P}^s$ implies $\Pi \models^c \mathsf{P}^c$.*

Proof. Let $\mathcal{A}$ be an arbitrary p.p.t. adversary for Π. We have

$$\Pr\left[\mathsf{Exec}_{\Pi(R_\Pi),\mathcal{A}(R_\mathcal{A})}(\eta) \in \mathsf{P}^c\right] \geq \Pr\left[\mathsf{Exec}_{\Pi(R_\Pi),\mathcal{A}(R_\mathcal{A})}(\eta) \in \mathsf{P}^c \wedge \exists t \in \mathsf{Exec}^s(\Pi), t \preceq \mathsf{Exec}_{\Pi(R_\Pi),\mathcal{A}(R_\mathcal{A})}(\eta)\right].$$

Since $\Pi \models^s \mathsf{P}^s$ and $\mathsf{concrete}(\mathsf{P}^s) \subseteq \mathsf{P}^c$ it follows that:

$$\Pr\left[\mathsf{Exec}_{\Pi(R_\Pi),\mathcal{A}(R_\mathcal{A})}(\eta) \in \mathsf{P}^c\right] \geq \Pr\left[\exists t \in \mathsf{Exec}^s(\Pi) \mid t \preceq \mathsf{Exec}_{\Pi(R_\Pi),\mathcal{A}(R_\mathcal{A})}(\eta)\right].$$

By Lemma 1, we deduce $\Pr\left[\mathsf{Exec}_{\Pi(R_\Pi),\mathcal{A}(R_\mathcal{A})}(\eta) \notin \mathsf{P}^c\right] \leq \nu_\mathcal{A}(\eta)$, *i.e.* $\Pi \models^{\mathsf{c}} \Pi^{\mathsf{c}}$. □

5.3 Secrecy Properties

In the symbolic model, secrecy is naturally expressed as a trace property: a message is secret if it cannot be derived by the adversary. In the computational model however, typical definitions are much stronger and they usually say that an attacker cannot obtain not only the secret, but also *any* partial information about the secret. In this section we give symbolic and computational definitions for the secrecy of nonces used in a protocol and prove a soundness theorem: if a nonce is deemed secret using symbolic techniques, then the nonce is secret with respect to the stronger, computational definition.

We concentrate on the case of secrecy of nonces since there is no canonical definition for secrecy of composed messages in the computational world. In addition, as noticed in [12], the definition of secrecy for keys for example has to be weaker than indistinguishability as soon as the encrypted messages contain some redundancy. However, if the keys are not used, then security of keys is similar to the security of nonces and our results yield meaningful results for symmetric key exchange.

SECRECY IN THE SYMBOLIC MODEL. Let Π be an arbitrary k-party protocol. We say that Π guarantees the *secrecy* of the nonce $X^j_{A_i} \in \mathsf{X}_n(A_i)$ if in all possible executions, each instantiation of this variable remains unknown to the adversary. Formally, this means that for every valid trace $(\mathsf{sid}_1, f_1, H_1), \ldots, (\mathsf{sid}_n, f_n, H_n)$ of the protocol, for every session id $\mathsf{sid}_p = (r, i, (a_1, \ldots, a_k))$ where $a_1, \ldots, a_k$ are honest agents (*i.e.* none of them appears in the **corrupt** query), we have $H_n \not\vdash n(a_i, j, r)$. If this is the case, we write $\Pi \models^s \mathsf{SecNonce}(i, j)$.

SECRECY IN THE COMPUTATIONAL MODEL. We define the secrecy of the nonce $X^j_{A_i}$ in protocol Π using an experiment $\mathbf{Exp}^{\mathrm{sec\_}b}_{\mathsf{Exec}_{\Pi,\mathcal{A}}}(i, j)(\eta)$ that we describe below. The experiment is parametrized by a bit b and involves an adversary $\mathcal{A}$. The input to the experiment is a security parameter η. It starts by generating two random nonces n_0 and n_1 in $\mathcal{C}^\eta.n$. Then the adversary $\mathcal{A}$ starts interacting with the protocol Π as in the experiment $\mathsf{Exec}_{\Pi,\mathcal{A}}(\eta)$: it generates new sessions, sends messages and receives messages to and from these sessions (as prescribed by the protocol). At some point in the execution the adversary initiates a session s in which the role of A_i is executed, and declares this session under attack. Then, in this session the variable $X^j_{A_i}$ is instantiated with n_b (*i.e.* one of the two nonces chosen in the beginning of the experiment, the selection being made according to the bit b). The rest of the execution is exactly as in $\mathsf{Exec}_{\Pi,\mathcal{A}}$. In the end, the adversary is given n_0 and n_1 and outputs a guess d, which is also the result of the experiment. We define the advantage of the adversary $\mathcal{A}$ by:

$$\mathbf{Adv}^{\mathrm{sec}}_{\mathsf{Exec}_{\Pi,\mathcal{A}}}(i, j)(\eta) = \Pr\left[\mathbf{Exp}^{\mathrm{sec\_}1}_{\mathsf{Exec}_{\Pi,\mathcal{A}}}(i, j)(\eta) = 1\right] - \Pr\left[\mathbf{Exp}^{\mathrm{sec\_}0}_{\mathsf{Exec}_{\Pi,\mathcal{A}}}(i, j)(\eta) = 1\right]$$

We say that nonce $X^j_{A_i}$ is computationally secret in protocol Π, and we write $\Pi \models^{\mathsf{c}} \mathsf{SecNonce}(i, j)$ if for every p.p.t. adversary $\mathcal{A}$ its advantage is negligible.

Our second main result, captured by the following theorem, states that if a nonce is secret in the symbolic model then it is also secret in the computational model.

Theorem 2. *Let Π be an executable protocol. If the schemes $\mathcal{DS}$ and $\mathcal{AE}$ are jointly secure, then: $\Pi \models^f \mathsf{SecNonce}(i, j)$ implies $\Pi \models^c \mathsf{SecNonce}(i, j)$.*

6 Automated Proof Using Casrul

In this section we describe the automated tool Casrul [9] and discuss the implications of our results for the proofs done with Casrul.

Casrul is a system for automated verification of cryptographic protocols, developed by the Cassis group at Loria (France) available at

`http://www.loria.fr/equipes/cassis/softwares/casrul/`

It translates a protocol given in common abstract syntax into a rewrite system. The rewrite system is processed using a first order theorem prover for equational logic for the automated detection of flaws. We note that Casrul does not allow the use of signatures and labels yet. Nevertheless, both its syntax and semantics coincide with ours for *public key protocols*, *i.e.* protocols that only use pairing and asymmetric encryption, but without using labels. We believe that both labels and signatures could be easily added in Casrul.

AUTOMATED PROOF FOR COMPUTATIONAL SECURITY USING CASRUL. Casrul can be used to prove three particular types of properties: entity authentication, authentication on data and data secrecy. Here, we discuss the implications of these proofs with respect to the computational model.

The syntax of Casrul does not yet allow the use of labels for encryption. However, it can be shown that for the security properties that are typically proved with Casrul, proofs in the execution model without labels are sound w.r.t. the model where labels are used. Thus, thanks to Theorem 1, Casrul proofs of the security with respect to these properties have a clear computational interpretation. For example, the Casrul proof that the Needham-Schroeder-Lowe [15] protocol is a secure mutual authentication protocol (file `NSPK_LOWE3.hlpsl`) implies the same property, but in the computational model.

Similarly, Casrul proofs of nonce secrecy imply, via Theorem 2, the strong, computational secrecy notion that we gave in Section 5.3. For example, Casrul enables to prove the computational secrecy of nonces used in the corrected Needham-Schroeder-Lowe protocol [15] (file `NSPK_LOWE2.hlpsl`) and in the SPLICE protocol [22] (file `SPLICE2.hlpsl`).

Note that Casrul works only with a finite number of sessions, thus proofs in the computational model are obtained only for that fixed number of sessions. Nevertheless, since our proofs consider adversaries that create an unbounded of sessions, we could also obtain proofs of computational security properties by using tools dedicated to an unbounded number of sessions like Hermes [8] or Securify [10]. This would require to first prove that protocols secure in the symbolic models of Securify or Hermes are also secure in our symbolic model. We believe this to be true since their symbolic models are very similar to ours. We did not use these tools for our proofs since they only provide automatic proofs of secrecy. Automated proofs of other security properties like authentication are still under development.

Acknowledgements. We would like to thank Martin Abadi for enlightening discussions and advice, Daniele Micciancio for useful suggestions and to the anonymous referees for their helpful remarks.

References

1. M. Abadi. Taming the adversary. In *Proc. of Crypto'00*, 2000.
2. M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proc. of the 4th Conf. on Computer and Communications Security*, pages 36–47. ACM Press, 1997.
3. M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.
4. M. Backes. Personal communication.
5. M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations (extended abstract). In *Proc. of 10th ACM Conference on Computer and Communications Security (CCS'05)*, pages 220 – 230, 2003.
6. M. Bellare, A. Boldyreva, and S. Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In *Proc. of Eurocrypt'00*, volume 1807 of *LNCS*, pages 259–274, 2000.
7. B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proc. of the 14th CSFW*, June 2001.
8. L. Bozga, Y. Lakhnech, and M. Perin. An automatic tool for the verification of secrecy in security protocols. In *15th Int. Conference on Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*, pages 219–222. Springer, July 2003.
9. Y. Chevalier and L. Vigneron. A tool for lazy verification of security protocols. In *Proc. of the 16th Conf. on Automated Software Engineering (ASE-2001)*. IEEE CS Press, 2001.
10. V. Cortier. *A guide for Securify*. RNTL EVA project, Report n. 13, December 2003.
11. V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. Research Report RR-5341, INRIA, October 2004.
12. D.H.Phan and D. Pointcheval. Une comparaison entre deux méthodes de preuve de sécurité. In *Proc. of RIVF*, pages 105–110, 2003. In French.
13. S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17(2):281–308, April 1988.
14. P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *Proc. of 2004 IEEE Symposium on Security and Privacy*, pages 71–85, 2004.
15. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *LNCS*, pages 147–166. Springer-Verlag, March 1996.
16. G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. of 10th CSFW'97*. IEEE Computer Society Press, 1997.
17. D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Theory of Cryptography Conference (TCC 2004)*, pages 133–151, Cambridge, MA, USA, February 2004. Springer-Verlag.
18. J. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague. A probabilistic polynomial-time calculus for analysis of cryptographic protocols. *Electronic Notes in Theoretical Computer Science*, 45, 2001.
19. L. Paulson. Mechanized proofs for a recursive authentication protocol. In *Proc. of the 10th CSFW'97*, pages 84–95. IEEE Computer Society Press, 1997.
20. C. Rackoff and D. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *CRYPTO'91*, pages 433–444, 1992.
21. M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. In *Proc. of the 14th CSFW'01*, pages 174–190. IEEE Computer Society Press, 2001.
22. S. Yamaguchi, K. Okayama, and H. Miyahara. The design and implementation of an authentication system for the wide area distributed environment. *IEICE Transactions on Information and Systems*, November 1991.

Completing the Picture: Soundness of Formal Encryption in the Presence of Active Adversaries

(Extended Abstract)

Romain Janvier, Yassine Lakhnech, and Laurent Mazaré

VERIMAG - 2, av. de Vignates, 38610 Gières - France
{romain.janvier, yassine.lakhnech, laurent.mazare}@imag.fr

Abstract. In this paper, we extend previous results relating the Dolev-Yao model and the computational model. We add the possibility to exchange keys and consider cryptographic primitives such as signature. This work can be applied to check protocols in the computational model by using automatic verification tools in the formal model.

To obtain this result, we introduce a precise definition for security criteria which leads to a nice reduction theorem. The reduction theorem is of interest on its own as it seems to be a powerful tool for proving equivalences between security criteria. Also, the proof of this theorem uses original ideas that seem to be applicable in other situations.

Note: An extended version of this paper appears as technical report [17].

1 Introduction

There are two approaches to the verification of cryptographic protocols. The so-called *formal approach* [1] that originates from the work of Dolev and Yao and was first described in [8]. The distinguishing feature of this approach is the perfect cryptography hypothesis that essentially postulates that an intruder can only gain information from an encoded message if he knows the inverse key. The other hypothesis is that fresh nonce creation is perfect. Even under these assumptions flaws have been found in protocols that were believed to be secure. Several automatic tools (whether complete in the case of bounded protocols or incomplete in the case of unbounded ones) have been developed (e.g., [18, 6, 11, 4]). In the second approach, encryption schemes are studied using a computational model based on Turing machines. In this context, there is no idealization made concerning the cryptographic schemes: cryptographic functions operate on strings, attackers are Turing machines and correctness is defined in terms of high complexity and weak probability of success [9, 3]. This computational approach is

[1] Formal is not used here in the sense of rigorous but denotes the use of formal methods.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 172–185, 2005.

recognized as more realistic than the formal approach. However, its complexity makes it very difficult to develop (semi-)automatic verification methods.

Therefore, a major research goal is to relate both approaches such that a protocol that is verified within the formal approach is guaranteed to be correct in the computational (that is without making the perfect cryptography hypothesis). This research has been initiated by the work of Abadi and Rogaway [2] and a later work [1] where it has been proved that a notion of indistinguishability in the formal model is valid in the computational model. This work has been pushed further in [19] and then in [15] where an active intruder is considered. This last paper proves that if the encryption scheme verifies a certain property (called IND-CCA), then security in the formal model implies security in the computational model. The important part of this work is that it applies to active adversaries. Other related works include Backes, Pfizmann and Waidner [14] where the formal model is not exactly the Dolev-Yao model, although very close. It is not clear whether protocols can be checked automatically in this formalism. Also, the cryptographic primitives are modeled at a rather detailed level in the computational model. P. Laud [12] proves safety of the formal model for symmetric encryption. In particular, he deals with encryption cycles.

Our objective in this paper is to continue this work and weaken some of the restrictions imposed on protocols in previous works. The main restriction in [15] is that secret keys cannot be part of sent messages and that message forwarding is not allowed. To weaken these restrictions, we first give a general definition of a security criterion (like IND-CCA). These criteria can be seen as a game that an intruder should not be able to win. Our first result is a reduction theorem that proves the equivalence between a criterion and simpler criteria. This allows us to prove that the IND-CCA criterion is equivalent to quite richer and useful criteria. Definition of criteria is an important part as they make it possible to release some of the restrictions over protocols made by previous works. These criteria are equivalent to IND-CCA, a well studied notion in provable cryptography. Finally, we use these criteria in order to prove that Dolev-Yao constitutes a safe abstraction of the computational model even for protocols involving both asymmetric encoding and digital signature.

2 Preliminaries

An *asymmetric encryption scheme* $\mathcal{AE} = (\mathcal{KG}, \mathcal{E}, \mathcal{D})$ is defined by three algorithms. The key generation algorithm $\mathcal{KG}$ is a randomized function which given a security parameter η outputs a pair of keys (pk, sk), where pk is a public key and sk the associated secret key. The encryption algorithm $\mathcal{E}$ is also a randomized function which given a message and a public key outputs the encryption of the message by the public key. Finally the decryption algorithm $\mathcal{D}$ takes as input a cyphertext and a secret key and outputs the corresponding plain-text, i.e., $\mathcal{D}(\mathcal{E}(m, pk), sk) = m$. The execution time of the three algorithms is assumed polynomially bounded by η.

A *signature scheme* $\mathcal{SS} = (\mathcal{KG}, \mathcal{S}, \mathcal{V})$ is also defined by three algorithms. The key generation algorithm randomly generates pairs of keys (sik, vk), where sik is the signature key and vk is the verification key. The signature algorithm $\mathcal{S}$ randomly produces a signature of a given message by a given signature key. The verification algorithm $\mathcal{V}$ is given a message m, a signature σ and a verification key vk and tests if σ is a signature of m with the signature key corresponding to vk. Hence, $\mathcal{V}(m, \mathcal{S}(m, sik), vk)$ returns true for any message m and any pair of keys (sik, vk) generated by $\mathcal{KG}$. In this case, we still assume that the algorithms have a polynomial complexity.

An adversary for a given scheme is a Polynomial Random Turing Machine (PRTM) which has access to a set of oracles. These oracles depend on the scheme and are given in the different cases thereafter. In the following, we consider Turing machines which execution is polynomially bounded in the security parameter η, i.e. for any input corresponding to security parameter η, the machine stops within $P(\eta)$ steps for some polynomial P.

To model access to oracles, we slightly modify the definition of Turing machines. Our Turing machines have two additional tapes, one for arguments (of function/oracle calls) and one for results. Then, let F be a countable set of function names. We define our PRTM as a pair of a Turing machine $\mathcal{A}$, where transitions can be function calls, and a substitution σ linking function names $\underline{f} \in F$ to functions from string of bits (arguments) to string of bits (results). These functions are also described by polynomial Turing machines (which can also access oracles). To distinguish oracles from real functions (which can be their implementations), function names are always underlined when considering access to an oracle. The semantics of $\mathcal{A}/\sigma$ are the standard semantics of $\mathcal{A}$ except that whenever $\mathcal{A}$ fires a transition labeled by a function call $\underline{f}$, the content of the results tape becomes $f\sigma(args)$, where $args$ is the value of the arguments tape.

To simplify notations, we write directly $\mathcal{A}/f_1, ..., f_n$ where f_i are functions. Thus, we omit the name of the function as soon as this name is not relevant for comprehension and functions are directly called using the $\underline{f_i}$ notation when defining $\mathcal{A}$.

A function $h : \mathbb{R} \to \mathbb{R}$ is *negligible*, if it is ultimately bounded by x^{-c}, for each positive $c \in \mathrm{N}$, i.e., for all $c > 0$ there exists N_c such that $|h(x)| < x^{-c}$, for all $x > N_c$.

The definition of messages and of the intruder in the formal model is by now standard, e.g. [8, 18].

3 A Generic Reduction Theorem

In [15], protocols allowing sending of secret keys are not considered because it is not possible in IND-CCA to encode secret keys. To solve that, we introduce a new criterion N-PAT-IND-CCA and prove it equivalent to IND-CCA. A similar result is needed to introduce signature.

A security criterion γ is defined by an experiment that involves an adversary and two ways W_0 and W_1 of implementing a set of oracles. The adversary is aware of both implementations and is allowed to call the oracles but does not know *a priori* which implementation is really used. The challenge consists in guessing which implementation is used. More precisely, an adversary is a probabilistic polynomial time Turing machine (PRTM) that has access to a set of oracles (either W_0 or W_1). The adversary's *advantage* is the probability that the adversary outputs 1 when the set of oracles is W_1 minus the probability that the adversary outputs 1 when the set of oracles is W_0. An encryption scheme is said γ-secure, if the advantage of any adversary is negligible.

In this section, we present a generic result allowing us to prove that a security criterion γ_1 can be reduced to a criterion γ_2. This means that if there exists an adversary that breaks γ_2 then there exists an adversary that breaks γ_1. The proof is constructive in the sense that such an adversary for γ_1 can be effectively computed.

Given a finite set x_i where i ranges from 1 to n, $\overline{x}$ denotes the whole set of x_i. When more precision is required, this set can also be denoted by $x_{1..n}$. In this section, we give a formal definition of a criterion and show how a criterion can be partitioned in a safe way. The theorem presented here allows us to verify that a criterion is equivalent to another one by using such partitions. This result is applied in the following sections to show an equivalence between a few security criteria.

3.1 Security Criterion

A criterion γ is a collection formed by:

- A bit b, this bit is the challenge that has to be guessed by the adversary.
- A finite number of parameters c_1 to c_{na}. These parameters are shared by the oracles and most of the time, they are chosen randomly at the beginning of the experiment. Θ is the PRTM producing these parameters (usually a key generation algorithm).
- A finite number of oracles f_1 to f_{nb} that depend on their argument, $\overline{c}$ and b. For each f_i, there exist f_i^α and f_i^β such that the corresponding oracles when given argument (l, r) produce $f_i^\beta(f_i^\alpha(l, \overline{c}), \overline{c})$ when $b = 0$ and $f_i^\beta(f_i^\alpha(r, \overline{c}), \overline{c})$ when $b = 1$.
- A finite number of oracles g_1 to g_{nc} that depend on their argument and $\overline{c}$. The corresponding oracles when given argument x produce $g_i(x, \overline{c})$.

Oracles in $\overline{g}$ do not depend on b, they cannot be used directly by the adversary to gain information on b but they can be useful by giving information on the shared parameters $\overline{c}$ that can finally allow the adversary to deduce the value of b. Oracles in $\overline{f}$ have two layers α and β, these layers are used to decompose a criterion into a partition of criteria as the α layer allows the β layers to depend on less parameters.

Example 1. Let γ be the criterion $(b, \{pk_1, sk_1, pk_2, sk_2\}, \{f_1, f_2\}, \emptyset)$. Functions f_1 and f_2 have no α layer, i.e. $f_i^\alpha(x, ...) = x$ and $f_i(m_0, m_1)$ corresponds to the

encryption of message m_b using key pk_i. Thus γ corresponds to 2-IND-CPA as introduced in [5]: the adversary has to guess the value of bit b by using only two oracles that encrypt the left or the right message according to b. The 2-IND-CCA criterion can be obtained by adding two oracles g_1 and g_2. These oracles decrypt messages encoded respectively with key pk_1 and pk_2 assuming that these messages have not been produced by oracle f_1 or f_2. □

The advantage of a PRTM $\mathcal{A}$ against γ is

$$\mathbf{Adv}^{\gamma}_{\mathcal{A}}(\eta) = Pr[\mathbf{Exp}^{\gamma}_{\mathcal{A}}(\eta, 1) = 1] - Pr[\mathbf{Exp}^{\gamma}_{\mathcal{A}}(\eta, 0) = 1]$$

Where *Exp* is the Turing machine defined by:

Experiment $\mathbf{Exp}^{\gamma}_{\mathcal{A}}(\eta, b)$:
$\overline{c} \xleftarrow{R} \Theta(\eta)$
if $b = 0$ **then**
$f_i \leftarrow \lambda(l, r).f_i^{\beta}(f_i^{\alpha}(l, \overline{c}), \overline{c})$ for i in $1...nb$
else
$f_i \leftarrow \lambda(l, r).f_i^{\beta}(f_i^{\alpha}(r, \overline{c}), \overline{c})$ for i in $1...nb$
$d \xleftarrow{R} \mathcal{A}/\eta, \overline{f}, \overline{g}$
return d

$\mathcal{A}$ has access to an oracle giving η and to the oracles $\overline{f}$ and $\overline{g}$ as defined above. Oracles in $\overline{f}$ depend on b, this dependence is explicited by "creating" oracles f according to the value of b.

The advantage of $\mathcal{A}$ is the probability to answer 1 when the value of b is 1 minus the probability to answer 1 when the value of b is 0. Thus, if we consider a machine $\mathcal{A}$ that always outputs the same result, its advantage is 0, this also holds when considering a machine that gives a random output. Advantages are between -1 and 1, however, if $\mathcal{A}$ has a negative advantage, it is easy to build a PRTM $\mathcal{B}$ that has the opposite of $\mathcal{A}$'s advantage (we simply need to run $\mathcal{A}$ and to return the inverse of its output).

3.2 Criterion Partition and the Reduction Theorem

Example 2. Let us consider the 2-IND-CPA criterion γ defined before. Then, we say that $\gamma' = (b, \{pk_1, sk_1\}, \{f_1\}, \emptyset)$ and $\gamma'' = (b, \{pk_2, sk_2\}, \{f_2\}, \emptyset)$ constitutes a valid partition of γ when both criteria are valid (i.e. f_1 and f_2 are in the same criterion as their respective parameters pk_1 and pk_2). γ' and γ'' correspond to the IND-CPA criterion (only one oracle is available). By the reduction theorem, if an encryption scheme is IND-CPA secure (advantage of any PRTM against γ' and γ'' is negligible), then it is 2-IND-CPA secure. □

A pair of criteria γ', γ'' defines a *valid partition* of γ if there exist na', nb' and nc' such that

- $\gamma' = \left(b, c_{1..na'}, f^{\beta}_{1..nb'}, g_{1..nc'}\right)$
- $\gamma'' = \left(b, c_{(na'+1)..na}, f_{(nb'+1)..nb}, g_{(nc'+1)..nc}\right)$

- For $i \leq nb'$, f_i^{α} only depends on $c_{(na'+1)..na}$.
- For $i \leq nb'$, f_i^{β} only depends on $c_{1..na'}$.
- For $i \leq nc'$, g_i only depends on $c_{1..na'}$.
- For $i > nb'$, f_i only depends on $c_{(na'+1)..na}$.
- For $i > nc'$, g_i only depends on $c_{(na'+1)..na}$.

The four last conditions are necessary for γ' and γ'' to remain valid: oracles from a criterion only have access to parameters generated by the same criterion. The reduction theorem states that an advantage against a criterion γ can be used to produce an advantage over criterion γ' or criterion γ''.

Theorem 1 (Reduction Theorem). *If γ', γ'' is a valid partition of γ and $\mathcal{A}$ is a PRTM, then there exist two PRTM $\mathcal{A}^o$ and $\mathcal{B}$ such that*

$$|\mathbf{Adv}_{\mathcal{A}}^{\gamma}(\eta)| \leq 2.|\mathbf{Adv}_{\mathcal{B}}^{\gamma'}(\eta)| + |\mathbf{Adv}_{\mathcal{A}^o}^{\gamma''}(\eta)|$$

Proof Idea for the Reduction Theorem. The purpose of this section is to explain the main ideas underlying the proof of the reduction theorem, the detailed proof appears in [17]. An application of this proof to a simple example is given below.

The adversary $\mathcal{A}^o$ against the criterion γ'' simulates $\mathcal{A}$. To do so, he has to answer the queries made to oracles from γ'. Since $\mathcal{A}^o$ cannot construct faithfully these oracles (as it does not have access to parameters from γ''), it returns incorrect answers to $\mathcal{A}$. Finally $\mathcal{A}^o$ uses the output of $\mathcal{A}$ to answer its own challenge. If the advantage of $\mathcal{A}^o$ is comparable to the advantage of $\mathcal{A}$, then γ can be reduced to criterion γ''.

Else the advantage of $\mathcal{A}^o$ is negligible compared to the advantage of $\mathcal{A}$, then another adversary, $\mathcal{B}$, has the same advantage as $\mathcal{A}$. The adversary $\mathcal{B}$ is playing against the criterion γ'. It generates a challenge for $\mathcal{A}$. Moreover, if $b = 1$ the answers to the queries made by $\mathcal{A}$ are correct and if $b = 0$ the answers are forged in the same way as in $\mathcal{A}^o$. When $\mathcal{A}$ answers its challenge, $\mathcal{B}$ verifies it. If it is correct, $\mathcal{B}$ supposes that $b = 1$, else it supposes that $b = 0$. Indeed, $\mathcal{A}^o$ probably has a lower advantage than $\mathcal{A}$.

Example 3. Consider our previous (IND-CPA) example. Machine $\mathcal{A}^o$ is opposed to γ'', it creates the missing key pk_1 and uses it to simulate the missing oracle: the simulation is achieved by always encoding the left argument, ${}_{fake}f_1(l, r) = \mathcal{E}(l, pk_1)$. Machine $\mathcal{B}$ is opposed to γ' with the challenge bit b'. It creates its missing key pk_2 and a random bit b. The fake oracle ${}_{fake}f_2$ uses this bit b. Oracle f_1 is also faked using b': ${}_{fake}f_1(m_0, m_1) = f_1(m_0, m_b)$. The faked oracles behave like the original oracles when $b' = 1$. They behave like the oracles faked in $\mathcal{A}^o$ when $b' = 0$. This behavior is summed up in the following array:

oracles	$b' = 0$	$b' = 1$
$\underline{\mathcal{E}_{pk_1}}(m_0, m_1)$	$\mathcal{E}(m_0, pk_1)$	$\mathcal{E}(m_b, pk_1)$
$\underline{\mathcal{E}_{pk_2}}(m_0, m_1)$	$\mathcal{E}(m_b, pk_2)$	$\mathcal{E}(m_b, pk_2)$

Then, if the underlying $\mathcal{A}$ machine answers b correctly, we assume that it was confronted to the right oracles and thus machine $\mathcal{B}$ answers 1, else it answers 0. The intuition behind machine $\mathcal{B}$ is that its advantage tells whether oracles from γ' are useful or can be faked without losing the advantage. □

4 Applications of the Reduction Theorem

We now introduce a new security criterion and prove it equivalent to IND-CCA by using our reduction theorem. N-PAT-IND-CCA allows the adversary to obtain the encryption of messages containing challenge secret keys, even if it does not know the value of these secret keys. For that purpose, the adversary is allowed to give pattern terms to the left-right oracles.

The pattern terms are terms where new atomic constants have been added: pattern variables. These variables denote the different challenge secret keys ($[i]$ asks the oracle to replace it with the value of sk_i). Variables can be used as atomic messages (data pattern). When a left-right oracle is given a pattern term, it replaces patterns by values of corresponding keys and encodes the message. More formally, patterns are given by the following grammar where bs is a bit-string and i is an integer.

$$pat ::= \langle pat, pat\rangle | \{pat\}_{bs} | bs | [i]$$

The computation (valuation) made by the oracle is easily defined recursively in a context giving the bit-string values for the different keys. Its result is a bit-string and it uses the encryption algorithm $\mathcal{E}$ and the concatenation denoted by operator $\cdot$.

$$v(bs, \overline{pk, sk}) = bs \qquad v(\{p\}_{bs}, \overline{pk, sk}) = \mathcal{E}(v(p, \overline{pk, sk}), bs))$$
$$v([i], \overline{pk, sk}) = sk_i \qquad v(\langle p_1, p_2\rangle, \overline{pk, sk}) = v(p_1, \overline{pk, sk}).v(p_2, \overline{pk, sk})$$

There is yet a restriction: we exclude encryption cycles. Hence keys are ordered and a pattern $[i]$ can only be encrypted under pk_j if $i > j$. References concerning this restriction appear in [2].

The related criterion is γ_N where $\overline{c}$ is a list containing N pairs of keys (pk_i, sk_i). Oracles in $\overline{f}$ are the encryption oracles. They behave like the oracles defined in the previous example except that they perform the replacement of pattern variables with key values. The v operation is performed in the α layer whereas the β layer corresponds to the previous oracles (i.e. simple encoding). Formally, $f_i^{\alpha}(x) = v(x, \overline{c})$ and $f_i^{\beta}(x) = \mathcal{E}(x, sk_i)$. Oracles in $\overline{g}$ decrypt a message using secret keys as long as their argument has not been produced by an oracle in $\overline{f}$.

An asymmetric encryption scheme $\mathcal{AE}$ is said to be N-PAT-IND-CCA iff for any adversary $\mathcal{A}$, $\mathbf{Adv}^{\gamma_N}_{\mathcal{AE},\mathcal{A}}(\eta)$ is negligible. Note that 1-PAT-IND-CCA corresponds to IND-CCA.

Lemma 1. *Let $\mathcal{AE}$ be an asymmetric encryption scheme. If $\mathcal{AE}$ is N-PAT-IND-CCA, then $\mathcal{AE}$ is also IND-CCA.*

Proposition 1. *Let $\mathcal{AE}$ be an asymmetric encryption scheme. If $\mathcal{AE}$ is N-PAT-IND-CCA, then $\mathcal{AE}$ is also (N+1)-PAT-IND-CCA.*

Proof. We have $c_i = (pk_i, sk_i)$. Then let γ' and γ'' be the partitions obtained with $na' = nb' = nc' = 1$; f_1^β and g_1^β only need c_1; f_1^α only needs $c_{2..(N+1)}$, this would not hold if we release the acyclicity hypothesis. Finally, f_i and g_i with $i \geq 2$ only need $c_{2..(N+1)}$ and so this is a valid partition. Hence, criterion γ_{N+1} has a valid partition constituted by γ_1 and γ_N.

The reduction theorem applies and gives:

$$|\mathbf{Adv}_{\mathcal{A}}^{\gamma_{N+1}}(\eta)| \leq 2.|\mathbf{Adv}_{\mathcal{B}}^{\gamma_1}(\eta)| + |\mathbf{Adv}_{\mathcal{A}^o}^{\gamma_N}(\eta)|$$

By hypothesis, $\mathcal{AE}$ is N-PAT-IND-CCA (hence IND-CCA). Then advantages of $\mathcal{B}$ and $\mathcal{A}^o$ are negligible and we can conclude that the advantage of $\mathcal{A}$ is negligible too. □

Corollary 1. *For any N, $\mathcal{AE}$ is IND-CCA if and only if $\mathcal{AE}$ is also N-PAT-IND-CCA.*

This result tells us that if an encryption scheme is IND-CCA secure, then it is still secure when adding the possibility to ask for encryption of patterns instead of just encryption of messages.

4.1 Signature

In order to extend previous results to the case of protocols using signature, we present here a new definition of security for signature scheme, UNF-CCA, which is an adaptation of Selective (Un)Forgery Against Adaptive Chosen Message Attack [10].

The main requirement is that an adversary should not be able to forge a pair containing a message m and the signature of m using the secret signature key. An N-UNF-CCA adversary $\mathcal{A}$ is given N verification keys and has to produce a message and its signature under one of the keys. It has access to the security parameter η, N verification keys vk_i and N signature oracles $\mathcal{S}_{sik_i}(.)$. The experiment outputs bit 1 if $\mathcal{A}$ managed to produce a compromising pair $(m, \{m\}_{sik_i})$ which right part is not the result of a call to a signature oracle. Otherwise, the experiment outputs bit 0. Formally the experiment is detailed below.

Experiment $\mathbf{Exp}_{\mathcal{SS},\mathcal{A}}^{N-UNF}(\eta)$:
for $i = 1$ **to** N **do**
$(sik_i, vk_i) \overset{R}{\leftarrow} \mathcal{KG}(\eta)$
$(m, \sigma) \overset{R}{\leftarrow} \mathcal{A}/\eta, vk_1, ..., vk_N,$
$\mathcal{S}_{sik_1}(.), ..., \mathcal{S}_{sik_N}(.),$
if σ is a valid signature of m under one of the sik_i
not produced by $\mathcal{S}_{sik_i}(.)$
return 1
else return 0

The advantage of adversary $\mathcal{A}$ in winning the UNF-CCA challenge is defined as:

$$\mathbf{Adv}^{N-UNF}_{\mathcal{SS},\mathcal{A}}(\eta) = Pr[\mathbf{Exp}^{N-UNF}_{\mathcal{SS},\mathcal{A}}(\eta) = 1]$$

A signature scheme $\mathcal{SS}$ is said to be N-UNF-CCA iff for any adversary $\mathcal{A}$, $\mathbf{Adv}^{N-UNF}_{\mathcal{SS},\mathcal{A}}(\eta)$ is negligible. Instead of 1-UNF-CCA, we write UNF-CCA.

As the challenge is not anymore guessing the value of a bit b, our reduction theorem cannot apply directly. However, by modifying the proof scheme given above, it is possible to deduce the following property relating the UNF-CCA and N-UNF-CCA criteria. The proof is given in [17].

Proposition 2. *For any signature scheme $\mathcal{SS}$, if $\mathcal{SS}$ is UNF-CCA, then it is also N-UNF-CCA.*

4.2 N-PAT-UNF-IND-CCA

To be able to deal with protocols using both an encryption scheme and a signature scheme, we define a new criterion N-PAT-UNF-CCA, a combination of N-PAT-IND-CCA and N-UNF-CCA. Let us consider an asymmetric encryption scheme $\mathcal{AE} = (\mathcal{KG}, \mathcal{E}, \mathcal{D})$ and a signature scheme $\mathcal{SS} = (\mathcal{KG}', \mathcal{S}, \mathcal{V})$. We use two types of adversary: those who try to find the secret bit b used in the N left-right pattern encryption oracles and those who try to produce a message and its signature under one of the N challenge signature keys. Each of these corresponds to an experiment, we denote by $N-PUI_1$ and $N-PUI_2$, respectively. The left-right pattern encryption oracles accept patterns of the form $[sik_i]$ where sik_i is one of the challenge signature keys. Then, the corresponding advantages are:

$$\mathbf{Adv}^{N-PUI_1}_{(\mathcal{AE},\mathcal{SS}),\mathcal{A}}(\eta) = Pr[\mathbf{Exp}^{N-PUI_1}_{(\mathcal{AE},\mathcal{SS}),\mathcal{A}}(\eta,1) = 1] - Pr[\mathbf{Exp}^{N-PU_1}_{(\mathcal{AE},\mathcal{SS}),\mathcal{A}}(\eta,0) = 1]$$

$$\mathbf{Adv}^{N-PUI_2}_{(\mathcal{AE},\mathcal{SS}),\mathcal{A}}(\eta) = Pr[\mathbf{Exp}^{N-PUI_2}_{(\mathcal{AE},\mathcal{SS}),\mathcal{A}}(\eta) = 1]$$

A couple $(\mathcal{AE}, \mathcal{SS})$ is said to be N-PAT-UNF-IND-CCA iff for all adversary $\mathcal{A}$, $\mathbf{Adv}^{N-PUI_1}_{(\mathcal{AE},\mathcal{SS}),\mathcal{A}}(\eta)$ and $\mathbf{Adv}^{N-PUI_2}_{(\mathcal{AE},\mathcal{SS}),\mathcal{A}}(\eta)$ are negligible.

The following property states that the combination of a secure signature scheme and a secure encryption scheme is still secure. Its proof can be done using the same proof scheme as for the reduction theorem.

Proposition 3. *If $\mathcal{AE}$ is N-PAT-IND-CCA and if $\mathcal{SS}$ is N-UNF-CCA, $(\mathcal{AE}, \mathcal{SS})$ is N-PAT-UNF-IND-CCA.*

To sum up, we proved the following equivalences between criteria:

$$\begin{gathered}\text{IND} - \text{CCA} \Leftrightarrow \text{N} - \text{PAT} - \text{IND} - \text{CCA}\\ \text{UNF} - \text{CCA} \Leftrightarrow \text{N} - \text{UNF} - \text{CCA}\\ (\text{IND} - \text{CCA}, \text{UNF} - \text{CCA}) \Leftrightarrow \text{N} - \text{PAT} - \text{UNF} - \text{IND} - \text{CCA}\end{gathered}$$

5 Dolev-Yao Is a Safe Abstraction

In this section, we give a precise formalization of the link between the two commonly used approaches for verification of cryptographic protocols, i.e. the computational approach and the formal approach. For that purpose, we first define cryptographic protocols, then we relate traces from both models. This relation is used to prove the main theorem.

5.1 Description of Cryptographic Protocols

A multi-party protocol is defined by a list of triples (m_1, m_2, R), called actions. The action (m_1, m_2, R) means that an agent playing role R sends a message m_2 after receiving a message m_1. It is possible to replace m_1 with an empty message denoted by "-" for the first action of the protocol. For the last action, the same thing can be done with m_2. A role R represents a program that is executed by an agent Ag during a session S. In session S, we say that agent Ag impersonates role R. Let us consider the Needham-Schroeder-Lowe protocol (NSL introduced in [13]) there are two roles: the initiator and the receiver.

For a session S, an agent Ag has some local variables used during the execution of his role: his local copy of variable Var is denoted by $Ag.S.Var$. Each agent has five variables for each of his sessions: $Init, Rec, Na, Nb, Pc$. The list of actions is:

- $(-, \{\langle Init, Na\rangle\}_{Pk_{Rec}}, Init)$
- $(\{\langle init, na\rangle\}_{Pk_{Rec}}, \{\langle Rec, na, Nb\rangle\}_{Pk_{init}}, Rec)$
- $(\{\langle Rec, Na, nb\rangle\}_{Pk_{Init}}, (\{nb\}_{Pk_{Rec}}, Init)$
- $((\{Nb\}_{Pk_{Rec}}, -, Rec)$

We make a distinction between values known before an action and values received during the action: values already known are denoted using a capital for their first letter. Let Ag be an agent impersonating the initiator. In the first action, Ag chooses B as a receiver, and the value of nonce Na for session s. He sets variables $Ag.s.Init$ to Ag, $Ag.s.Rec$ to B and $Ag.s.Na$ with the chosen value. Then he sends message $\{\langle Ag.s.Init, Ag.s.Na\rangle\}_{Ag.s.Rec}$. In the second action, B receives a message encrypted with his public key. He decrypts it and parses the plain-text as a pair $\langle init, na\rangle$. After that, he chooses a new session number s' and sets $B.s'.Init$ to $init$, $B.s'.Rec$ to B, $B.s'.Na$ to na and finally chooses a fresh value for $B.s'.Nb$. Finally, he sends message $\{\langle B.s'.Rec, B.s'.Na, B.s'.Nb\rangle\}_{Pk_{B.s'.Init}}$ to Ag. The remaining actions have similar semantics.

Hypothesis over Protocols. The following restrictions are made over the protocol Π considered in this section. Π has to be executable, that is each role can be run by a PRTM. In the formal world, for any execution, secret keys of honest agents remain secrets, there is no encryption cycle, there is a nonce in each signed message that also remains secret. Moreover, we ask that messages include enough typing information to allow parsing of received messages. We also assume that any agent knows identities of all the other agents.

Computational and Formal Models. For both models, agents involved in a protocol send their messages through a network. This network is modeled by the Adversary. The Adversary intercepts any message sent by an agent. He can forge new messages using his knowledge and send these messages to agents usurping an agent's identity.

In the formal model, the Adversary is a classical Dolev-Yao Intruder [8]. In the computational model, both the Adversary and the implementation of the protocol are PRTM, denoted by $\mathcal{A}_c$ and Π_c. Π_c is used as an oracle by $\mathcal{A}_c$. Messages are bit-strings, new nonces generated by agents are random bit-strings. Keys used by agents are generated using $\mathcal{KG}$. Encryptions and decryptions are obtained using algorithms from $\mathcal{AE}$. Signatures related functions use $\mathcal{SS}$.

$\mathcal{A}_c$ can create new valid identities and thus impersonate some dishonest agents.

5.2 Non Dolev-Yao Traces

A trace is a list of tuples (m_1, m_2, Ag, s) called transitions where m_1 is a message sent by the Adversary to agent Ag for session s and m_2 is the answer from Ag. As before, message m_1 and m_2 can be "-". Assignment $t_c \leftarrow \mathcal{A}_c/\eta, \Pi_c$ denotes that the trace t_c is obtained by the computational Adversary $\mathcal{A}_c$ confronted to Π_c . We assume that only messages accepted by an agent appear in the trace. We now transform a computational trace into a *pseudo formal trace*. The resulting trace is only pseudo formal because even if messages are expressed using Dolev-Yao terms, this does not imply that there exists a formal Adversary producing this trace. The transformation given here can be seen as verification of the trace by all the honest agents working together. Their goal is to check if the adversary performed an action that is not in the Dolev-Yao model. To achieve this, messages in the computational trace (which are bit-strings) are replaced with Dolev-Yao terms using the following:

- Bit-strings corresponding to identities are associated to fresh atoms: H_1, $H_2, ...$ for honest agents and $I_1, I_2, ...$ for dishonest agents.
- Bit-strings corresponding to long-term keys are associated to fresh keys: $Pk_{H_1}, Pk_{H_2}, ...$ and $Pk_{I_1}, Pk_{I_2}, ...$ for public keys, Sk_{Id} for associated secret keys.
- Bit-strings corresponding to nonces N generated in session S by an honest agent are associated to fresh atoms N^s.
- Bit-strings corresponding to fresh public or secret keys generated by an honest agent in session s are associated with fresh keys Pk^s and Sk^s.
- Bit-strings corresponding to keys generated by the Adversary are associated with fresh keys Pk_I^j and Sk_I^j.
- Bit-strings corresponding to concatenation of a message associated to term T_1 with a message associated to term T_2 are associated with $\langle T_1, T_2 \rangle$.
- Bit-strings corresponding to encryption of messages associated to term T with a key associated to term K are associated to term $\{T\}_K$.

Note that this is not complete as we have not yet taken into account nonces generated by the Adversary. As the Adversary is a PRTM, whenever he has to

send a new nonce, he does not have to generate it randomly: he can send composed messages instead of nonces or perform operations over bit-strings (XOR, changing bit order, adding or removing bits...). Hence for the honest agents it is impossible to guess how the Adversary has chosen his nonces. This is why when transforming a bit-string corresponding to a message where an honest agent receives a new nonce, we only test if the corresponding bit-string is an already known nonce. In this case, the bit-string is associated to the nonce term. Else, it is associated to a fresh variable X_i. If later this bit-string is parsed as something else (tuple, encoding), variable X_i is replaced by the appropriate term. The same thing is done when an honest agent receives a message encrypted with a key which inverse is not known or a signature impossible to verify (at reception time). When every message in the trace has been transformed, each remaining X_j is replaced by the fresh atom N_I^j, i.e. remaining variables are considered as fresh nonces. The pseudo-formal trace corresponding to computational trace t_c is denoted by $\alpha(t_c)$.

Definition 1 (Non Dolev-Yao Traces). *A formal trace t_f is said* Non Dolev-Yao *(NDY) iff there exists a message sent by the Adversary which cannot be deduced from previous messages using Dolev-Yao's deduction, this message is called a NDY message. A computational trace t_c is said NDY iff $\alpha(t_c)$ is NDY.*

5.3 A Computational Trace Is Certainly a Dolev-Yao Trace

In this section, we prove that if the encryption and signature schemes verify IND-CCA resp. UNF-CCA and if the number of possible nonces is exponential in η, then the probability that a computational trace is NDY is negligible. This means that the computational Adversary, even with all the computing power of PRTM, cannot have a behavior not represented by a formal adversary.

Theorem 2. *Let Π be a protocol. Let $\mathcal{AE}$ be the encryption scheme and $\mathcal{SS}$ the signature scheme used in Π_c. If $\mathcal{AE}$ is IND-CCA and $\mathcal{SS}$ is UNF-CCA then for any concrete Adversary $\mathcal{A}_c$:*
$Pr\big[t_c \leftarrow \mathcal{A}_c/\eta, \Pi_c\ ;\ t_c\ NDY\big]$ is negligible.

The proof of this theorem can be found in [17].

5.4 Formal and Computational Properties

Let P_c be a property in the computational world represented by a predicate over computational traces. A protocol Π verifies P_c (denoted by $\Pi \models_c P_c$) iff for any Adversary $\mathcal{A}_c$, $Pr\big[t_c \leftarrow \mathcal{A}_c/\eta, \Pi_c\ ;\ \neg P_c(t_c)\big]$ is negligible. A property in the formal world is represented by a predicate P_f over formal traces. Hence, a protocol Π verifies P_f (denoted by $\Pi \models_f P_f$) iff any trace produced by a Dolev-Yao adversary against Π verifies P_f.

Using theorem 2, we prove the following result which states that proving formally P_f allows us to deduce P_c.

Theorem 3. *Let P_f and P_c be a formal and a computational property such that*

$$\forall t_c, \forall t_f, \big(P_f(t_f) \wedge \alpha(t_c) = t_f\big) \Rightarrow P_c(t_c)$$

If $\mathcal{AE}$ is IND-CCA and $\mathcal{SS}$ is UNF-CCA, then

$$\Pi \models_f P_f \Rightarrow \Pi \models_c P_c$$

This theorem states that if the formal property correctly under-approximates the computational property then the formal abstraction is correct.

This theorem has been applied to mutual authentification in [15] and holds for nonce secrecy [7].

6 Conclusion

In this paper, we considered active intruders. Our main result is that an adversary behavior follows the formal model with overwhelming probability, if the encryption scheme is IND-CCA and the signature scheme is UNF-CCA. This result has immediate applications as automatic verification of security protocols is quite developed now and as there are encryption algorithms that verify the required properties. Our result extends previous ones and allow:

- Multi-party protocols.
- More cryptographic primitives: combination of digital signature and asymmetric encryption.
- Protocols where encoding of secret keys and message forwarding are allowed.

A second main contribution of our paper is a formal definition for security criteria and a reduction theorem. This theorem and its proof scheme seem to apply in a wide variety of cases. It allows to prove equivalences between a security criterion and some of its sub-criteria. This theorem allowed us to give a quick proof of already known results, to generalize this to new results and we believe that it could be useful whenever one wants to relate two security criteria.

Concerning extensions of this work, in [16], we extend these results to protocols using simultaneously all the classical cryptographic primitives: asymmetric and symmetric encoding, signature and hashing. This paper also deals with simple equational theories.

Acknowledgments

This work has been partially supported by the RNTL project PROUVE-03V360 and the ACI project ROSSIGNOL

References

1. M. Abadi and J. Jürgens. Formal eavesdropping and its computational interpretation. In *4th Intern. Symp. on Theoretical Aspects of Computer Software*, volume 2215 of *LNCS*. Springer-Verlag, 2001.
2. Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *IFIP International Conference on Theoretical Computer Science (IFIP TCS2000)*, Sendai, Japan, 2000. Springer-Verlag, Berlin Germany.

3. Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of cipher block chaining. In Yvo G. Desmedt, editor, *Proc. CRYPTO 94*, pages 341–358. Springer, 1994. Lecture Notes in Computer Science No. 839.
4. B. Blanchet. Abstracting cryptographic protocols by prolog rules. In *International Static Analysis Symposium*, volume 2126 of *LNCS*, pages 433–436, 2001.
5. A. Boldyreva, M. Bellare, and S. Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In *Advances in Cryptology-EUROCRYPT 2000, Lecture Notes in Comput. Sci., Vol. 1807*, pages 259–274. Springer, 2000.
6. L. Bozga, Y. Lakhnech, and M. Périn. Hermes: An automatic tool for verification of secrecy in security protocols. In *15th International Conference on Computer Aided Verification (CAV),*, volume 2725 of *LNCS*, 2003.
7. V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. Research Report RR-5341, INRIA, October 2004.
8. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
9. S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, April 1984.
10. Shafi Goldwasser, Silvio Micali, and Ron L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.
11. Jean Goubault-Larrecq. A method for automatic cryptographic protocol verification. In *International Workshop on Formal Methods for Parallel Programming: Theory and Applications*, volume 1800 of *LNCS*, 2000.
12. P. Laud. Symmetric encryption in automatic analyses for confidentiality against adaptive adversaries. In *Proc. of 2004 IEEE Symposium on Security and Privacy*, pages 71–85, 2004.
13. G. Lowe. An attack on the Needham-Schroeder public-key authentification protocol. *Information Processing Letters*, 56(3):131–133, 1995.
14. B. Pfitzmann M. Backes and M. Waidner. U universally composable cryptographic library. In ACM Press, editor, *Computer and Communication Security*, Oct. 2003.
15. D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Proceedings of the Theory of Cryptography Conference*, pages 133–151. Springer, 2004.
16. Y. Lakhnech R. Janvier and L. Mazaré. (de)compositions of cryptographic schemes and their applications to protocols. Technical report, Verimag, Centre Équation, 38610 Gières, To Appear 2004.
17. Y. Lakhnech et L. Mazare R. Janvier. Completing the picture: Soundness of formal encryption in the presence of active adversaries. Technical Report TR-2004-19, Verimag, Centre Équation, 38610 Gières, November 2004.
18. M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. In *IEEE Computer Security Foundations Workshop*, 2001.
19. B. Warinschi. A computational analysis of the needham-schroeder(-lowe) protocol. In *Proceedings of 16th Computer Science Foundation Workshop*, pages 248–262. ACM Press, 2003.

Analysis of an Electronic Voting Protocol in the Applied Pi Calculus

Steve Kremer[1] and Mark Ryan[2]

[1] Laboratoire Spécification et Vérification,
CNRS UMR 8643 & INRIA Futurs projet SECSI & ENS Cachan, France
kremer@lsv.ens-cachan.fr
[2] School of Computer Science,
University of Birmingham, UK
M.D.Ryan@cs.bham.ac.uk

Abstract. Electronic voting promises the possibility of a convenient, efficient and secure facility for recording and tallying votes in an election. Recently highlighted inadequacies of implemented systems have demonstrated the importance of formally verifying the underlying voting protocols. The applied pi calculus is a formalism for modelling such protocols, and allows us to verify properties by using automatic tools, and to rely on manual proof techniques for cases that automatic tools are unable to handle. We model a known protocol for elections known as FOO 92 in the applied pi calculus, and we formalise three of its expected properties, namely fairness, eligibility, and privacy. We use the ProVerif tool to prove that the first two properties are satisfied. In the case of the third property, ProVerif is unable to prove it directly, because its ability to prove observational equivalence between processes is not complete. We provide a manual proof of the required equivalence.

1 Introduction

Electronic voting promises the possibility of a convenient, efficient and secure facility for recording and tallying votes. It can be used for a variety of types of elections, from small committees or on-line communities through to full-scale national elections. However, the electronic voting machines used in recent US elections have been fraught with problems. Recent work [13] has analysed the source code of the machines sold by the second largest and fastest-growing vendor, which are in use in 37 US states. This analysis has produced a catalogue of vulnerabilities and possible attacks.

A potentially much more secure system could be implemented, based on formal protocols that specify the messages sent between the voters and administrators. Such protocols have been studied for several decades. They offer the possibility of abstract analysis of the protocol against formally-stated properties. There are two main kinds of protocol proposed for electronic voting [16]. In blind signature schemes, the voter first obtains a token, which is a message blindly signed by the administrator and known only to the voter herself. She later sends her vote anonymously, with this token as proof of eligibility. In schemes using homomorphic encryption, the voter cooperates with the

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 186–200, 2005.

administrator in order to construct an encryption of her vote. The administrator then exploits homomorphic properties of the encryption algorithm to compute the encrypted tally directly from the encrypted votes.

Among the properties which electronic voting protocols may satisfy are the following:

Fairness: no early results can be obtained which could influence the remaining voters.
Eligibility: only legitimate voters can vote, and only once.
Privacy: the fact that a particular voted in a particular way is not revealed to anyone.
Individual verifiability: a voter can verify that her vote was really counted.
Universal verifiability: the published outcome really is the sum of all the votes.
Receipt-freeness: a voter cannot prove that she voted in a certain way (this is important to protect voters from coercion).

In this paper, we study a protocol commonly known as the FOO 92 scheme [12], which works with blind signatures. By informal analysis (e.g., [16]), it has been concluded that FOO 92 satisfies the first four properties in the list above.

Because security protocols are notoriously difficult to design and analyse, formal verification techniques are particularly important. In several cases, protocols which were thought to be correct for several years have, by means of formal verification techniques, been discovered to have major flaws [14, 6]. Our aim in this paper is to use verification techniques to analyse the FOO 92 protocol. We model it in the applied pi calculus [3], which has the advantages of being based on well-understood concepts. The applied pi calculus has a family of proof techniques which we can use, is supported by the ProVerif tool [4], and has been used to analyse a variety of security protocols [1, 11].

2 The FOO 92 Protocol

The protocol involves voters, an administrator, verifying that only eligible voters can cast votes, and a collector, collecting and publishing the votes. In comparison with authentication protocols, the protocol also uses some unusual cryptographic primitives, such as secure bit-commitment and blind signatures. Moreover, it relies on anonymous channels.

In a first phase, the voter gets a signature on a commitment to his vote from the administrator. To ensure privacy, blind signatures [7] are used, i.e. the administrator does not learn the commitment of the vote.

- Voter V selects a vote v and computes the commitment $x = \xi(v,r)$ using the commitment scheme ξ and a random key r;
- V computes the message $e = \chi(x, b)$ using a blinding function χ and a random blinding factor b;
- V digitally signs e and sends his signature $\sigma_V(e)$ to the administrator A together with his identity;
- A verifies that V has the right to vote, has not voted yet and that the signature is valid; if all these tests hold, A digitally signs e and sends his signature $\sigma_A(e)$ to V;
- V now *unblinds* $\sigma_A(e)$ and obtains $y = \sigma_A(x)$, i.e. a signed commitment to V's vote.

The second phase of the protocol is the actual voting phase.

- V sends y, A's signature on the commitment to V's vote, to the collector C using an anonymous channel;
- C checks correctness of the signature y and, if the test succeeds, enters (ℓ, x, y) onto a list as an ℓ-th item.

The last phase of the voting protocol starts, once the collector decides that he received all votes, e. g. after a fixed deadline. In this phase the voters reveal the random key r which allows C to open the votes and publish them.

- C publishes the list (ℓ_i, x_i, y_i) of commitments he obtained;
- V verifies that his commitment is in the list and sends ℓ, r to C via an anonymous channel;
- C opens the ℓ-th ballot using the random r and publishes the vote v.

Note that we need to separate the voting phase into a commitment phase and an opening phase to avoid releasing partial results of the election.

3 The Applied Pi Calculus

The applied pi calculus [3] is a language for describing concurrent processes and their interactions. It is based on the pi calculus, but is intended to be less pure and therefore more convenient to use. Properties of processes described in the applied pi calculus can be proved by employing manual techniques [3], or by automated tools such as ProVerif [4]. As well as reachability properties which are typical of model checking tools, ProVerif can in some cases prove that processes are observationally equivalent [5]. This capability is important for privacy-type properties such as those we study here. The applied pi calculus has been used to study a variety of security protocols, such as those for private authentication [11] and for fast key establishment [1].

To describe processes in the applied pi calculus, one starts with a set of *names* (which are used to name communication channels or other constants), a set of *variables*, and a *signature* Σ which consists of the function symbols which will be used to define terms.

In the applied pi calculus, one has (plain) processes and extended processes. Plain processes are built up in a similar way to processes in the pi calculus, except that messages can contain terms (rather than just names). Extended processes can also be *active substitutions*: $\{^M/_x\}$ is the substitution that replaces the variable x with the term M. Active substitutions generalise "let". The process $\nu x.(\{^M/_x\} \mid P)$ corresponds exactly to "let $x = M$ in P".

Active substitutions are useful because they allow us to map an extended process A to its *frame* $\phi(A)$ by replacing every plain processes in A with 0. A frame is an extended process built up from 0 and active substitutions by parallel composition and restriction. The frame $\phi(A)$ can be viewed as an approximation of A that accounts for the static knowledge A exposes to its environment, but not A's dynamic behaviour.

The operational semantics of processes in the applied pi calculus is defined by structural rules defining two relations: *structural equivalence*, noted $\equiv$, and *internal reduction*, noted $\rightarrow$. A context $C[\cdot]$ is a process with a hole; an evaluation context is a context

whose hole is not under a replication, a conditional, an input, or an output. Structural equivalence is is the smallest equivalence relation on extended processes that is closed under α-conversion on names and variables, by application of evaluation contexts, and satisfying some further basic structural rules such as $A \mid 0 \equiv A$, associativity and commutativity of $|$, binding-operator-like behaviour of ν, and when $\Sigma \vdash M = N$ the equivalences:

$$\nu x.\{^M/_x\} \equiv 0 \qquad \{^M/_x\} \mid A \equiv \{^M/_x\} \mid A\{^M/_x\} \qquad \{^M/_x\} \equiv \{^N/_x\}$$

Internal reduction $\rightarrow$ is the smallest relation on extended processes closed under structural equivalence such that $\bar{a}\langle x\rangle.P \mid a(x).Q \rightarrow P \mid Q$ and whenever $\Sigma \not\vdash M = N$,

$$\text{if } M = M \text{ then } P \text{ else } Q \rightarrow P \qquad \text{if } M = N \text{ then } P \text{ else } Q \rightarrow Q.$$

Many properties of security protocols (including some of the properties we study in this paper) are formalised in terms of *observational equivalence* between processes. To define this, we write $A \Downarrow a$ when A can send a message on a, that is, when $A \rightarrow^* C[\bar{a}\langle M\rangle.P]$ for some evaluation context C that does not bind a.

Definition 1. *Observational equivalence ($\approx$) is the largest symmetric relation R between closed extended processes with the same domain such that $A\ R\ B$ implies:*

1. *if $A \Downarrow a$ then $B \Downarrow a$.*
2. *if $A \rightarrow^* A'$ then $B \rightarrow^* B'$ and $A'\ R\ B'$ for some B'.*
3. *$C[A]\ R\ C[B]$ for closing evaluation contexts C.*

In cases in which the two processes differ only by the terms they contain, if they are also observationally equivalent then ProVerif may be able to prove it directly. However, ProVerif's ability to prove observational equivalence is incomplete, and therefore sometimes one has to resort to manual methods, whose justifications are contained in [3].

The method we use in this paper relies on two further notions: *static equivalence* ($\approx_s$), and *labelled bisimilarity* ($\approx_l$). Static equivalence just compares the static knowledge processes expose to their environment. Two frames are statically equivalent if, when considered as substitutions, they agree on the distinguishability of terms. For frames, static equivalence agrees with observational equivalence, while for general extended processes, observational equivalence is finer.

The definition of *labelled* bisimilarity is like the usual definition of bisimilarity, except that at each step in the unravelled definition one additionally requires that the processes are statically equivalent. Labelled bisimilarity and observational equivalence coincide [3]. Therefore, to prove observational equivalence, it is sufficient to prove bisimilarity and static equivalence at each step. This is what we do to prove the privacy property.

4 Modelling FOO 92 in the Applied Pi Calculus

4.1 Model

We use the applied pi calculus to model the FOO 92 protocol. The advantage is that we can combine powerful (hand) proof techniques from the applied pi calculus with

```
(* Signature *)
fun commit/2     (* bit commitment *)
fun open/2       (* open bit commitment *)
fun sign/2       (* digital signature *)
fun checksign/2  (* open digital signature *)
fun pk/1         (* get public key from private key *)
fun host/1       (* get host from public key *)
fun getpk/1      (* get public key from host *)
fun blind/2      (* blinding *)
fun unblind/2    (* undo blinding *)

(* Equational theory *)
equation open(commit(m,r),r) = m
equation getpk(host(pubkey))=pubkey
equation checksign(sign(m,sk),pk(sk)) = m
equation unblind(blind(m,r),r) = m
equation unblind(sign(blind(m,r),sk),r) = sign(m,sk)
```

Process 1. Signature and equational theory

automated proofs provided by Blanchet's ProVerif tool. Moreover, the verification is not restricted to a bounded number of sessions and we do not need to explicitly define the adversary. We only give the equational theory describing the intruder theory. Generally, the intruder has access to any message sent on a public, i.e. unrestricted, channel. These public channels model the network. Note that all channels are anonymous in the applied pi calculus. Unless the identity or something like the IP address is specified explicitly in the conveyed message, the origin of a message is unknown. This abstraction of a real network is very appealing, as it avoids having us to model explicitly an anonymiser service. However, we stress that a real implementation needs to treat anonymous channels with care.

Most of our proofs rely directly on Blanchet's ProVerif tool. The input for the tool is given in an ascii version of the applied pi calculus. To be as precise as possible, the processes described below are directly extracted out of the input files and are given in a pretty-printed version of the ascii input. The minor changes with the usual applied pi calculus notation should be clear.

4.2 Signature and Equational Theory

The signature and equational theory are represented in Process 1. We model cryptography in a Dolev-Yao style as being perfect. In this model we can note that bit commitment (modeled by the functions commit and open) is identical to classical symmetric-key encryption. The functions and equations that handle public keys and hostnames should be clear. Digital signatures are modeled as being signatures with message recovery, i.e. the signature itself contains the signed message which can be extracted using the checksign function. To model blind signatures we add a pair of functions blind and unblind. These functions are again similar to perfect symmetric key encryption and bit commitment.

```
process
    ν ska. ν skv. (* private keys *)
    ν privCh. (* channel for registering legimitate voters *)
    let pka=pk(ska) in
    let hosta = host(pka) in
    let pkv=pk(skv) in
    let hostv=host(pkv) in
    (* publish host names and public keys *)
    out(ch,pka). out(ch,hosta).
    out(ch,pkv). out(ch,hostv).
    (* register legimitate voters *)
    ((out(privCh,pkv). out(privCh,pk(ski))) |
    (!processV)|(!processA)|(!processC))
```

Process 2. Environment process

However, we add a second equation which permits us to extract a signature out of a blinded signature, when the blinding factor is known. The ProVerif tool also implicitly handles pairing: pair(x,y) is abbreviated as (x,y). We also consider the functions fst and snd to extract the first, respectively second element of a pair. Note that because of the property that unblind(sign(blind(m,r),sk),r) = sign(unblind(blind(m,r),r),sk) = sign(m,sk), our theory is not a subterm theory. Therefore the results for deciding static equivalence from [2] do not apply. However, an extension of [2] presents new results that seem to cover a more general family of theories, including the one considered here [9].

4.3 The Environment Process

The main process is specified in Process 2. Here we model the environment and specify how the other processes (detailed below) are combined. First, fresh secret keys for the voters and the administrator are generated using the restriction operator. For simplicity, all legitimate voters share the same secret key in our model (and therefore the same public key). The public keys and hostnames corresponding to the secret keys are then sent on a public channels, i.e. they are made available to the intruder. The list of legitimate voters is modeled by sending the public key of the voters to the administrator on a private communication channel. We also register the intruder as being a legitimate voter by sending his public key pk(ski) where ski is a free variable: this enables the intruder to introduce votes of his choice and models that some voters may be corrupted. Then we combine an unbounded number of each of the processes (voter, administrator and collector). An unbounded number of administrators and collectors models that these processes are servers, creating a separate instance of the server process (e.g. by "forking") for each client.

4.4 The Voter Process

The voter process given in Process 3 models the role of a voter. At the beginning two fresh random numbers are generated for blinding, respectively bit commitment of the vote. Note that the vote is not modeled as a fresh nonce. This is because generally the

```
let processV =
    ν blinder. ν r.
    let blindedcommitedvote=blind(commit(v,r),blinder) in
    out(ch,(hostv,sign(blindedcommitedvote,skv))).
    in(ch,m2).
    let blindedcommitedvote0=checksign(m2,pka) in
    if blindedcommitedvote0=blindedcommitedvote then
    let signedcommitedvote=unblind(m2,blinder) in
    phase 1.
    out(ch,signedcommitedvote).
    in(ch,(l,=signedcommitedvote)).
    phase 2.
    out(ch,(l,r))
```

Process 3. Voter process

```
let processA =
    in(privCh,pubkv). (* register legimitate voters *)
    in(ch,m1).
    let (hv,sig)=m1 in
    let pubkeyv=getpk(hv) in
    if pubkeyv = pubkv then
    out(ch,sign(checksign(sig,pubkeyv),ska))
```

Process 4. Administrator process

domain of values of the votes are known. For instance this domain could be $\{yes, no\}$, a finite number of candidates, etc. Hence, vulnerability to guessing attacks is an important topic. We will discuss this issue in more detail in section 5. The remainder of the specification follows directly the informal description given in section 2. The command in(ch,(l,=s)) means the process inputs not any pair but a pair whose second argument is s. Note that we use phase separation commands, introduced by the ProVerif tool as global synchronization commands. The process first executes all instructions of a given phase before moving to the next phase. The separation of the protocol in phases is useful when analyzing fairness and the synchronization is even crucial for privacy to hold.

4.5 The Administrator Process

The administrator is modeled by the process represented in Process 4. In order to verify that a voter is a legitimate voter, the administrator first receives a public key on a private channel. Legitimate voters have been registered on this private channel in the environment process described above. The received public key has to match the voter who is trying to get a signed ballot from the administrator. If the public key indeed matches, then the administrator signs the received message which he supposes to be a blinded ballot.

```
let processC =
    phase 1.
    in(ch,m3).
    ν l. out(ch,(l,m3)).
    phase 2.
    in (ch,(=l,rand)).
    let voteV=open(checksign(m3,pka),rand) in
    out(ch,voteV)
```

Process 5. Collector process

4.6 The Collector Process

In Process 5 we model the collector. When the collector receives a committed vote, he associates a fresh label 'l' with this vote. Publishing the list of votes and labels is modeled by sending those values on a public channel. Then the voter can send back the random number which served as a key in the commitment scheme together with the label. The collector receives the key matching the label and opens the vote which he then publishes. Note that in this model the collector immediately publishes the vote without waiting that all voters have committed to their vote. In order to verify in section 5 that no early votes can be revealed we simply omit the last steps in the voter and collector process corresponding to the opening and publishing of the results.

5 Analysis

We have analysed three major properties of electronic voting protocols: fairness, eligibility and privacy. Most of the properties can be directly verified using ProVerif. The tool allows us to verify standard secrecy properties as well as resistance against guessing attacks, defined in terms of equivalences. For all but one property, privacy, the tool directly succeeds its proofs. When analysing privacy, we need to rely on the proof techniques introduced in [3]. Although the results are positive results, we believe that the way we verify the properties increases the understanding of the properties themselves and also the way to model them.

5.1 Fairness

Fairness is the property that ensures that no early results can be obtained and influence the vote. Of course, when we state that no early results can be obtained, we mean that the protocol does not leak any votes before the opening phase. It is impossible to prevent "exit polls", i.e. people revealing their vote when asked.

We model fairness as a secrecy property: it should be impossible for an attacker to learn a vote before the opening phase, i.e. before the beginning of phase 2.

Standard Secrecy. Checking *standard* secrecy, i.e. secrecy based on reachability, is the most basic property ProVerif can check. We request ProVerif to check that the private free variable v representing the vote cannot be deduced by the attacker. ProVerif directly succeeds to prove this result.

Resistance Against Guessing Attacks. In the previous paragraph we deduce that a *standard* attacker cannot learn a legitimate voter's vote. However, voting protocols are particularly vulnerable to *guessing attacks* because the values of the votes are taken from a small domain of possible values. Intuitively, in a guessing attack, an attacker *guesses* a possible value for the secret vote and then tries to verify his guess. A trivial example of a guessing attack is when the voter encrypts his vote with the collector's public key (using deterministic encryption). Then the attacker just needs to encrypt his guess and compare the result with the observed encrypted vote. Guessing attacks have been formalized by Lowe [15] and later by Delaune and Jacquemard [10]. A definition in terms of equivalences has been proposed by Corin et al. in [8]:

Definition 2. *Let ϕ be a frame in which v is free. Then we say that ϕ verifies a guess of v if $\phi \not\approx_s \nu v.\phi$. Conversely, we say that ϕ is secure wrt v if $\phi \approx_s \nu v.\phi$.*

Intuitively, if ϕ and $\nu v.\phi$ can be distinguished then an adversary can verify his guess using ϕ. This is also the definition checked by ProVerif. ProVerif succeeds in proving this stronger version of secrecy for the commitment phase of the FOO 92 protocol. Note that verification of guessing attacks does not support considering the protocol up to a given phase. Therefore, we slightly change the processes presented in section 4: we omit the last sending of the voter process which allows the opening of the commitment.

Strong Secrecy. We also verified *strong secrecy* in the sense of [5]. Intuitively, strong secrecy is verified if the intruder cannot distinguish between two processes where the secret changes. For the precise definition, we refer the reader to [5]. The main difference with guessing attacks is that strong secrecy relies on observational equivalence rather than static equivalence. ProVerif directly succeeds to prove strong secrecy.

Corrupt Administrator. We have also verified standard secrecy, resistance against guessing attacks and strong secrecy in the presence of a corrupt administrator. A corrupt administrator is modeled by outputting the administrator's secret key on a public channel. Hence, the intruder can perform any actions the administrator could have done. Again, the result is positive: the administrator cannot learn the votes of a honest voter, before the committed votes are opened. Note that we do not need to model a corrupt collector, as the collector never uses his secret key, i.e. the collector could anyway be replaced by the attacker.

5.2 Eligibility

Eligibility is the property verifying that only legitimate voters can vote, and only once. The way we verify the first part of this property is by giving the attacker a *challenge vote*. We modify the processes in two ways: (i) the attacker is not registered as a legitimate voter; (ii) the collector tests whether the received vote is the challenge vote and outputs the restricted name attack if the test succeeds. The modified collector process is given in Process 6. Verifying eligibility is now reduced to secrecy of the name attack. ProVerif succeeds in proving that attack cannot be deduced by the attacker.

If we register the attacker as a legitimate voter, the tool finds the trivial attack, where the intruder votes *challenge vote*. Similarly, if a corrupt administrator is modeled then the intruder can generate a signed commitment to the challenge vote and insert it.

```
let processC =
    phase 1.
    in(ch,m3).
    ν l. out(ch,(l,m3)).
    phase 2.
    in (ch,(=l,rand)).
    let voteV=open(checksign(m3,pka),rand) in
    ν attack.
        if voteV=challengeVote then
                out(ch, attack)
        else
                out(ch, voteV)
```

Process 6. Modified collector process for checking the eligibility properties

The second part of the eligibility property (that a voter can vote only once) cannot be verified in our model, because of our simplifying assumption that all voters share the same key.

5.3 Privacy

The privacy property aims to guarantee that the link between a given voter V and his vote v remains hidden. Anonymity and privacy properties have been successfully studied using equivalences. However, the definition of privacy in the context of voting protocols is rather subtle. While generally most security properties should hold against an arbitrary number of dishonest participants, arbitrary coalitions do not make sense here. Consider for instance the case where all but one voter are dishonest: as the results of the vote are published at the end, the dishonest voter can collude and determine the vote of the honest voter. A classical trick for modeling anonymity is to ask whether two processes, one in which V_1 votes and one in which V_2 votes, are equivalent. However, such an equivalence does not hold here as the voters' identities are revealed (and they need to be revealed at least to the administrator to verify eligibility). In a similar way, an equivalence of two processes where only the vote is changed does not hold, because the votes are published at the end of the protocol. To ensure privacy we need to hide the *link* between the voter and the vote and not the voter or the vote itself.

In order to give a reasonable definition of privacy, we need to suppose that at least two voters are honest. We denote the voters V_1 and V_2 and their votes $vote_1$, respectively $vote_2$. We say that a voting protocol respects privacy whenever a process where V_1 votes $vote_1$ and V_2 votes $vote_2$ is observationally equivalent to a process where V_1 votes $vote_2$ and V_2 votes $vote_1$.

With respect to the modeling given in section 4 we explicitly add a second voter. However, the equivalence that is checked by ProVerif is strictly finer than observational equivalence. Therefore the tool does not succeed in proving the above given privacy property. In Process 7, we illustrate a simple process that is observationally equivalent (it is actually structurally equivalent), but cannot be proven so by ProVerif. This example also illustrates ProVerif's choice operator used to define two processes that should be

```
process
    let x=choice[v1,v2] in
    let y=choice[v2,v1] in
    ( (out(ch,x)) | (out(ch,y)) )
```

Process 7. Limitation of the ProVerif tool to prove observational equivalence

proven observationally equivalent. The choice operator is a binary operator that defines two processes P_1 and P_2 such that choice(x_1,x_2) evaluates to x_1 in P_1 and to x_2 in P_2. Although the two processes are structurally equivalent, the current version of ProVerif does not succeed in proving observational equivalence.

As ProVerif takes as input processes in the applied pi calculus, we can rely on hand proof techniques to show privacy. The processes modeling the two voters are shown in Process 8. The main process is adapted accordingly to publish public keys and host names.

Proposition 1. *The FOO 92 protocol respects privacy, i.e.* $P[vote_1/v_1, vote_2/v_2] \approx P[vote_2/v_1, vote_1/v_2]$*, where* P *is given in Process 9.*

The proof can be sketched as follows. First note that the only difference between $P[vote_1/v_1, vote_2/v_2]$ and $P[vote_2/v_1, vote_1/v_2]$ lies in the two voter processes. We therefore first show that

$$\begin{gathered}(processV1|processV2)[vote_1/v_1, vote_2/v_2] \\ \approx \\ (processV1|processV2)[vote_2/v_1, vote_1/v_2].\end{gathered}$$

To prove this we show labelled bisimilarity. We denote the left-hand process as P_1 and the right-hand process as P_2. The labelled transition of P_1

$$\begin{aligned} P_1 \overset{\nu x1.\bar{ch}\langle x1\rangle}{\longrightarrow} & \nu blinder1.\nu r1.\nu blinder2.\nu r2. \\ & (P_1' | \{^{(hostv1, sign(blind(commit(v1,r1),blinder1),skv1)}/_{x1}\}) \\ \overset{\nu x2.\bar{ch}\langle x2\rangle}{\longrightarrow} & \nu blinder1.\nu r1.\nu blinder2.\nu r2. \\ & (P_1'' | \{^{(hostv1, sign(blind(commit(v1,r1),blinder1),skv1)}/_{x1}\} \\ & | \{^{(hostv2, sign(blind(commit(v2,r2),blinder1),skv2)}/_{x2}\}) \end{aligned}$$

can be simulated by P_2 as

$$\begin{aligned} P_2 \overset{\nu x1.\bar{ch}\langle x1\rangle}{\longrightarrow} & \nu blinder1.\nu r1.\nu blinder2.\nu r2. \\ & (P_2' | \{^{(hostv1, sign(blind(commit(v2,r1),blinder1),skv1)}/_{x1}\}) \\ \overset{\nu x2.\bar{ch}\langle x2\rangle}{\longrightarrow} & \nu blinder1.\nu r1.\nu blinder2.\nu r2. \\ & (P_2'' | \{^{(hostv1, sign(blind(commit(v2,r1),blinder1),skv1)}/_{x1}\} \\ & | \{^{(hostv2, sign(blind(commit(v1,r2),blinder1),skv2)}/_{x2}\}) \end{aligned}$$

For the first input of both voters, we need to consider two cases: either the input of both voters corresponds to the expected messages from the administrator or any other

```
(* Voter1 *)
let processV1 =
    ν blinder1. ν r1.
    let blindedcommitedvote1=blind(commit(v1,r1),blinder1) in
    out(ch,(hostv1,sign(blindedcommitedvote1,skv1))).
    in(ch,m21).
    let blindedcommitedvote01=checksign(m21,pka) in
    if blindedcommitedvote01=blindedcommitedvote1 then
    let signedcommitedvote1=unblind(m21,blinder1) in
    phase 1.
    out(ch,signedcommitedvote1).
    in(ch,(l1,=signedcommitedvote1)).
    phase 2.
    out(ch,(l1,r1))

(* Voter2 *)
let processV2 =
    ν blinder2. ν r2.
    let blindedcommitedvote2=blind(commit(v2,r2),blinder2) in
    out(ch,(hostv2,sign(blindedcommitedvote2,skv2))).
    in(ch,m22).
    let blindedcommitedvote02=checksign(m22,pka) in
    if blindedcommitedvote02=blindedcommitedvote2 then
    let signedcommitedvote2=unblind(m22,blinder2) in
    phase 1.
    out(ch,signedcommitedvote2).
    in(ch,(l2,=signedcommitedvote2)).
    phase 2.
    out(ch,(l2,r2))
```

Process 8. Two voters for checking the privacy property

message has been introduced by the attacker. In the first case, both voters synchronize at phase 1 and the frames of P_1, respectively P_2 are

$$\phi_1 = \nu blinder1.\nu r1.\nu blinder2.\nu r2.\{^{(hostv1,sign(blind(commit(v1,r1),blinder1),v1))}/_{x1}, {}^{(hostv2,sign(blind(commit(v2,r2),blinder2),v2))}/_{x2}, {}^{sign(blind(commit(v1,r1),blinder1),skva)}/_{x3}, {}^{sign(blind(commit(v2,r2),blinder2),skva)}/_{x4}\}$$

$$\phi_2 = \nu blinder1.\nu r1.\nu blinder2.\nu r2.\{^{(hostv1,sign(blind(commit(v2,r1),blinder1),v1))}/_{x1}, {}^{(hostv2,sign(blind(commit(v1,r2),blinder2),v2))}/_{x2}, {}^{sign(blind(commit(v2,r1),blinder1),skva)}/_{x3}, {}^{sign(blind(commit(v1,r2),blinder2),skva)}/_{x4}\}$$

```
process
    ν ska. ν skv1. ν skv2. (* private keys *)
    ν privCh. (* channel for registrating legimitate voters *)
    let pka=pk(ska) in
    let hosta = host(pka) in
    let pkv1=pk(skv1) in
    let hostv1=host(pkv1) in
    let pkv2=pk(skv2) in
    let hostv2=host(pkv2) in
    (* publish host names and public keys *)
    out(ch,pka). out(ch,hosta).
    out(ch,pkv1). out(ch,hostv1).
    out(ch,pkv2). out(ch,hostv2).
    let v1=choice[vote1,vote2] in
    let v2=choice[vote2,vote1] in
    ((out(privCh,pkv1). out(privCh,pkv2). out(privCh,pk(ski))) |
    (processV1)|(processV2)|(!processA)|(!processC))
```

Process 9. Main process with two voters

Given our equational theory and the fact that the blinding factors are restricted, these frames are statically equivalent. In the second case, if at least one input does not correspond to the correct administrator's signature, both voter processes will block, as testing correctness of the message fails and hence they cannot synchronize.

After the synchronization at phase 1, the remaining of the voter processes are structurally equivalent: the remaining of the first voter's process of P_1 is equivalent to the remaining of the second voter's process of P_2 and vice-versa. Due to this structural equivalence, P_2 can always simulate P_1 (and vice-versa). Moreover static equivalence will be ensured: with respect to frames ϕ_1 and ϕ_2 no other difference will be introduced and the blinding factors are never divulged.

Given observational equivalence of the voter processes, we can conclude observational equivalence of the the whole process, as observational equivalence is closed under application of closed evaluation contexts.

Note also that the use of phases is crucial for privacy to be respected. Surprisingly, when we omit the synchronization after the registration phase with the administrator, privacy is violated. Consider the following scenario. Voter 1 contacts the administrator. As no synchronization is considered, voter 1 can send his commited vote to the collector before voter 2 contacts the administrator. As voter 2 could not have submitted the commited vote, the attacker can link this commitment to the first voter's identity. This problem was found during a first attempt to prove the protocol where the phase instructions were omitted. The original paper divides the protocol into three phases but does not explain the crucial importance of the synchronization after the first phase. Our analysis emphasizes this need and we believe that it increases the understanding of some subtle details of the privacy property in this protocol.

6 Conclusion

We have modelled the FOO 92 electronic voting scheme in the applied pi calculus, and proved three kinds of property. Each property is checked either by reachability analysis or by checking observational equivalence:

Fairness. F1: the vote of a particular voter is not leaked to an attacker (reachability). F2: a guess of a vote cannot be verified by the attacker and strong secrecy is guaranteed (observational equivalence). These properties are also proved in the presence of a corrupt administrator.

Eligibility. E1: an attacker cannot trick the system into accepting his vote (reachability).

Privacy. P1: the attacker cannot distinguish the actual situation from one in which two voters have swapped their votes (observational equivalence).

The reachability properties (F1, E1) and the first observational equivalence property (F2) can be proved by ProVerif. The other observational equivalence property (P1) is more delecate, both in the way it is formulated and in the way that it is proved. ProVerif cannot prove this observational equivalence automatically. Therefore we proved it manually, by showing that the two processes are labelled-bisimilar.

In proving P1 manually, we noticed a feature of the protocol which is not much stressed in the descriptions (e.g. [12, 16]) but is vital for the proof: every participant must finish the registration stage before proceeding to the voting stage, and every participant must finish the voting stage before the collector can begin opening the votes. Otherwise, some attacks are possible. For example, if voting could begin before everyone has registered, the attacker could break privacy by temporarily blocking all registrations but V's. If V then votes, the attacker can establish a link between V and V's vote. We used the phase construct of ProVerif to prevent this.

Acknowledgments. Many thanks to Bruno Blanchet for suggestions about using ProVerif, as well as to Mathieu Baudet and Stéphanie Delaune for interesting discussions and comments. Thanks also to anonymous reviewers for helpful comments. Steve Kremer's work was partially supported by the RNTL project PROUVÉ and the ACI-SI Rossignol.

References

1. Martín Abadi, Bruno Blanchet, and Cedric Fournet. Just fast keying in the pi calculus. In David Schmidt, editor, *13th European Symposium on Programming (ESOP'04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 340–354, Barcelona, Spain, March 2004. Springer.
2. Martín Abadi and Véronique Cortier. Deciding knowledge in security protocols under equational theories. In Josep Diaz, Juhani Karhumäki, Arto Lepistö, and Don Sannella, editors, *31st Int. Coll. Automata, Languages, and Programming (ICALP'04)*, volume 3142 of *Lecture Notes in Computer Science*, pages 46–58, Turku, Finland, July 2004. Springer.
3. Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In Hanne Riis Nielson, editor, *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, pages 104–115, London, UK, January 2001. ACM.

4. Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In Steve Schneider, editor, *14th IEEE Computer Security Foundations Workshop*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society Press.
5. Bruno Blanchet. Automatic Proof of Strong Secrecy for Security Protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, Oakland, California, May 2004.
6. Rohit Chadha, Steve Kremer, and Andre Scedrov. Formal analysis of multi-party contract signing. In Riccardo Focardi, editor, *17th IEEE Computer Security Foundations Workshop*, pages 266–279, Asilomar, CA, USA, June 2004. IEEE Computer Society Press.
7. David Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology, Proceedings of CRYPTO'82*, pages 199–203. Plenum Press, 1983.
8. Ricardo Corin, Jeroen Doumen, and Sandro Etalle. Analysing password protocol security against off-line dictionary attacks. In *2nd International Workshop on Security Issues with Petri Nets and other Computational Models (WISP'04)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004. To appear.
9. Véronique Cortier. Personal communication, 2004.
10. Stéphanie Delaune and Florent Jacquemard. A theory of dictionary attacks and its complexity. In Riccardo Focardi, editor, *17th IEEE Computer Security Foundations Workshop*, pages 2–15, Asilomar, Pacific Grove, CA, USA, June 2004. IEEE Computer Society Press.
11. Cedric Fournet and Martin Abadi. Hiding names: Private authentication in the applied pi calculus. In *International Symposium on Software Security (ISSS'02)*, pages 317–338. Springer, 2003.
12. Atsushi Fujioka, Tatsuaki Okamoto, and Kazui Ohta. A practical secret voting scheme for large scale elections. In J. Seberry and Y. Zheng, editors, *Advances in Cryptology — AUSCRYPT '92*, volume 718 of *Lecture Notes in Computer Science*, pages 244–251. Springer, 1992.
13. Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2004.
14. Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.
15. Gavin Lowe. Analysing protocols subject to guessing attacks. In Joshua Guttman, editor, *In Proceedings of the Workshop on Issues in the Theory of Security (WITS'02)*, Portland, Oregon, USA, January 2002.
16. Zuzana Rjaskova. Electronic voting schemes. Master's thesis, Comenius University, 2002. `www.tcs.hut.fi/~helger/crypto/link/protocols/voting.html`.

Streams with a Bottom in Functional Languages

Hideki Tsuiki and Keiji Sugihara

Graduate School of Human and Environmental Studies,
Kyoto University, Kyoto, Japan
Fax: +81-75-753-6744
{tsuiki, sugihara}@i.h.kyoto-u.ac.jp

Abstract. When an infinite sequence contains a bottom cell, we cannot access the rest of the sequence with the ordinary stream access. On the other hand, when we consider an extended stream access with two heads, we can read or write $1\bot$-sequences, which are infinite sequences with at most one bottom cell. In this paper, we present a way of extending a lazy functional language with such an extended stream access in the realm of sequential computation. It has an application in real number computation in that the set of real numbers is topologically embedded in the set of $1\bot$-sequences [16], and therefore we can consider a program with such an extended stream access as directly manipulating real numbers. We implemented this mechanism by modifying the runtime of the Hugs system, which is a graph-reduction based implementation of the Haskell language. We present programming examples like addition and multiplication on real numbers in this extended Haskell.

For this implementation, we extended Haskell with the `gamb` operator, which works just as McCarthy's bottom-avoiding nondeterministic choice operator "amb". The difference is that it is realized in the realm of sequential computation, and that it is applicable only when the graph representations of the arguments share the same redex. In order to show that programs corresponding to two-head stream accesses satisfy this condition, we introduce a PCF-based calculus of term-graphs and define a data-type of $1\bot$-streams as a subtype of `[Bool]`.

1 Introduction

Stream is a useful data structure used in expressing, for example, process communication, and can be manipulated easily in functional languages. We are interested in boolean streams, so a stream in this paper means an infinite sequence of 0 and 1, which is accessed from left to right. One way of representing a stream in a functional language is to use the list type `[Bool]`. However, the type `[Bool]` includes infinite sequences with bottoms and if a program tries to make a stream access to input such an infinite sequence, it will be stuck at the bottom cell because the computation to obtain the value of the cell will not terminate. Therefore, with stream access, the part of the sequence after the first bottom is discarded, though the rest of the sequence may have valuable information.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 201–216, 2005.

The first author has found that streams with bottoms are useful in representing continuous topological spaces like $\mathcal{R}$, and performing computation over them. We will call an infinite sequence which may contain at most n copies of bottom an $n\bot$-sequence, and denote by $\Sigma^{\omega}_{\bot,n}$ the set of $n\bot$-sequences. It is shown in [15] that any n-dimensional separable metric space can be topologically embedded in $\Sigma^{\omega}_{\bot,n}$, and in particular, $\mathcal{R}$ can be topologically embedded in $\Sigma^{\omega}_{\bot,1}$ by what we call the Gray-code embedding. It means that each real number has a unique representation as a $1\bot$-sequence, and through this representation, the approximation structures of $\Sigma^{\omega}_{\bot,1}$ and $\mathcal{R}$ coincide. Note that $\mathcal{R}$ cannot be embedded in Σ^{ω} and therefore the existence of $\bot$ is essential; $\mathcal{R}$ is a 1-dimensional connected space whereas Σ^{ω} is a 0-dimensional totally disconnected space. Thus, if we have a computation which fills (or reads) a $1\bot$-sequence infinitely, then we can consider that it is outputting (or inputting) a real number. As such, he considered a machine called an IM2-machine. This machine has two heads on each input/output tape to make an extended stream access on $1\bot$-sequences. It is shown that the induced computability notion of the real functions coincides with the standard one [19].

In this paper, we present a way of extending a functional language with the two-head stream access of an IM2-machine in the realm of sequential computation. In [17], it is shown that we can express the behavior of an IM2-machine naturally with a logic programming language with guarded clauses and committed choice, such as Concurrent Prolog, PARLOG, and GHC (Guarded Horn Clauses). Therefore, we can already execute our real-number computation algorithms on ordinary computers. However, since what we are expressing are "functions" over the reals like addition and multiplication, it is more desirable that we can express them as functions in functional programming languages. In addition, if they are implemented in functional languages, we can also apply higher-order functions like "map" and "foldr" to real number functions, which is impossible with the above logic programming languages.

It is easy to show that the two-head stream access of an IM2-machine can be implemented if we consider parallel computation and use McCarthy's "amb" operator [11]. The amb operator is a bottom-avoiding nondeterministic choice operator, defined so that `amb` M N is reduced to V if M has the value V, V' if N has the value V', and its computation does not terminate only when both M and N do not have normal forms. Note that if the computations of both of the arguments are terminating, `amb` M N has two possibilities. Therefore, `amb` is a nondeterministic multi-valued function. In order to compute `amb` M N, we need to compute the values of M and N in parallel, and we can express the "parallel or" operator with "`amb`." There are some researches extending parallel functional languages with the amb operator [2]. However, such an implementation requires complicated control and scheduling over threads. Moreover, the motivation and goal of such parallel operator is different from ours. Nondeterminism and multi-valuedness are known to be essential in real-number computation, and, correspondingly, IM2-machines are defining nondeterministic multi-valued functions. However, IM2-machines are performing sequential computation, and

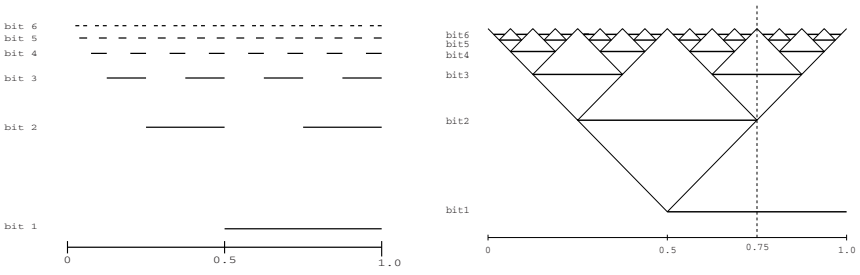


Fig. 1. The binary expansion and the Gray-code expansion of real numbers. Here, horizontal line means that the corresponding bit has value 1 [16]

real-number computation is not related with parallelism in this context. Therefore, it is more natural to implement it in the realm of sequential computation, and see what causes nondeterminism and multi-valuedness without using parallelism.

Thus, we consider our extension to sequential functional languages, and introduce a variant `gamb` of the `amb` operator to a graph-reduction based functional language. It is defined as an extension of the Haskell language and implemented by modifying the runtime system of the Hugs system, which is based on the notion of G-machines[7, 6, 8]. This implementation is available from the author's homepage[18], with some programming examples like addition and multiplication on real numbers.

Our `gamb` operator is a partial sequential realization of the `amb` operator. The difference is that `gamb` works sequentially, and that it works only for the case that the two term-graphs M and N given as the arguments share the same subgraph L as a redex and the normal form of one of M and N is composed from the weak head normal form of L only by list and boolean operations. In order to show that programs corresponding to two-head stream accesses satisfy this condition, we introduce a PCF-based calculus of term-graphs and define a datatype of $1\bot$-streams as a subtype of `[Bool]`. This datatype also brings out a set of primitive operators to manipulate $1\bot$-streams.

In Section 2, we overview Gray-code based real-number computation and IM2-machines, and show how it is expressed with McCarthy's "amb" operator. In Section 3, we introduce the operator `gamb` and explain how it works on term-graphs. In Section 4, we define $\mathrm{GPCF}^{\omega}_{\bot,1}$ and study its type system. In Section 5, we explain how `gamb` is implemented as an extension of Haskell. In Section 6, we explain programming examples of real number functions and higher order functions which use the `gamb` operator.

Notations: We consider the unit closed interval $\mathcal{I} = [0, 1]$ instead of the whole real line. We use 0 and 1 for the boolean values false and true, for simplicity. We fix the alphabet $\Sigma = \{0, 1\}$, and denote by Σ^{ω} the set of infinite sequences of Σ. We call an infinite sequence of $\{0, 1, \bot\}$ which may include at most one copy of $\bot$ an $1\bot$-sequence, and denote by $\Sigma^{\omega}_{\bot,1}$ the set of $1\bot$-sequences. Except for

section 4, we use the word term-graph informally for a graph representation of a (lambda) term.

2 Gray-Code and Real-Number Computation

2.1 Gray-Code Embedding

Gray-code expansion is an expansion of $\mathcal{I} = [0, 1]$ as infinite sequences of $\{0, 1\}$, which is different from the ordinary binary expansion. Figure 1 shows the binary and Gray-code expansion of $\mathcal{I}$. In the binary expansion of x, the head h of the expansion indicates whether x is in [0, 1/2] or [1/2, 1], and the tail is the expansion of $f(x, h)$ for f the following function:

$$f(x,h) = \begin{cases} 2x & (\text{when } h = 0) \\ 2x - 1 & (\text{when } h = 1) \end{cases}.$$

Note that the rest of the expansion depends on the choice of the head character h when $x = 1/2$. On the other hand, the head of the Gray-code expansion is the same as that of the binary expansion, whereas the tail is the expansion of $t(x)$ for t the so-called tent function:

$$t(x) = \begin{cases} 2x & (0 \le x \le 1/2) \\ 2(1-x) & (1/2 < x \le 1) \end{cases}.$$

This expansion is based on Gray code[5], which is a binary coding of natural numbers different from the ordinary one.

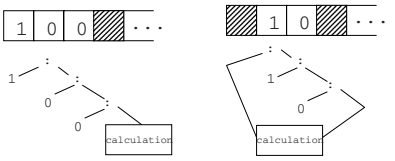
We have two binary expansions for a dyadic number (a rational number of the form $m/2^k$). For example, 3/4 has two expansions 110000... and 101111.... It is also the case for the Gray-code expansion, and 3/4 has two expansions 111000... and 101000.... Note that they differ only at one bit. It means that the second bit does not contribute to the fact that the value is 3/4, and it is more natural to leave it undefined ($\perp$). Thus, we define the expansion of 3/4 as $1\perp 1000\ldots$, and define the modified Gray-code expansion as follows.

Definition 1 ([16], [4]). Let $\Sigma = \{0, 1\}$ and $P : \mathcal{I} \to \Sigma_\perp$ be the map

$$P(x) = \begin{cases} 0 & (x < 1/2) \\ \perp & (x = 1/2) \\ 1 & (x > 1/2) \end{cases}.$$

Gray-code embedding G is a function from $\mathcal{I}$ to $\Sigma^\omega_{\perp,1}$ defined as $G(x)[n] = P(t^n(x))$ $(n = 0, 1, \ldots)$. We call $G(x)$ the *modified Gray-code expansion* of x.

Note that $\perp$ appears only once in each modified gray-code expansion. G is a topological embedding of $\mathcal{I}$ in $\Sigma^\omega_{\perp,1}$, where the topology of $\Sigma^\omega_{\perp,1}$ is given as the subspace topology of the Scott topology of ${\Sigma_\perp}^\omega$.



(a) Ordinary stream (b) 1⊥-stream

Fig. 2. The process of outputting streams

2.2 IM2-Machine

We study how the ordinary stream access can be extended to input/output 1⊥-sequences. We explain it with the way the Gray-code of a real number is input or output by a program.

For the output, we consider that a program (effectively) computes a real number x when it executes infinitely and produces better and better approximations of x as shrinking intervals. Therefore, we study how the Gray-code of x is output on a tape based on such information. When the information $x < 1/2$ or $1/2 < x$ is obtained by such a shrinking interval, a machine can write 0 or 1 on the first cell, respectively. However, when $x = 1/2$, neither information is given however long it waits and therefore it cannot fill the first cell forever. However, in this case, it obtains the information $1/4 < x < 3/4$ at some time, and it can write 1 on the second cell skipping the first one if it is allowed to write a character not only on the leftmost unfilled cell but also on the next unfilled cell. After that, if the information $1/4 < x < 1/2$ or $1/2 < x < 3/4$ is given, it can write 0 or 1 on the skipped cell, respectively, and if it has the information $3/8 < x < 5/8$, it can write 0 on the third (i.e., the second unfilled) cell. After that, the computation of x will have the possibility to fill the first or the 4th cell of the string as Figure 2 shows. In this way, when $x = 1/2$, the first cell is left unfilled and the sequence $1000\ldots$ is written from the second cell. Thus, if the output tape is filled with $\bot$ at the beginning, we can output the modified Gray-code expansion on the tape.

We can formulate this mechanism as an output with two heads. We consider two heads H_1 and H_2 on each tape. They are located at the first two cells at the beginning, and only H_2 moves to the next cell after an output from H_2, and H_1 moves to the position of H_2 and H_2 moves to the next cell after an output from H_1. In this way, the two heads are always located at the first and the second unfilled cell of the string. This is a generalization of the ordinary stream access with one head, which is located at the first unfilled cell and moves to the next cell after an output.

As for the input, when the value of a cell is $\bot$, a machine cannot wait for it to be filled. Therefore, in order to skip a bottom cell and continue the input, we need two heads also on input tapes, which move the same way as output-tape heads. Then, when both of the cells under the two heads are filled, a machine has two possible inputs which may cause two different computations. Therefore, it has nondeterministic behavior and both of the computational paths must produce valid results.

In this way, we define a machine, called an IM2-machine (Indeterministic Multi-head Type2-machine), which has two heads on each input/output tape and which has nondeterministic behavior depending on the order it inputs. See [16] for the detailed definition of an IM2-machine.

2.3 IM2-Machine Outputs with Functional Languages

Ordinary stream access can be expressed in a lazy functional language as a recursively defined function of type `[Bool]` → `[Bool]`. As for the stream access with two heads, it is trivially impossible to express it in functional languages because multi-valued functions are not definable in functional languages. It is also shown in [17] that some IM2-computable single valued functions are not expressible in functional languages when $1\bot$-sequences are implemented as `[Bool]`. Therefore, for such an implementation, we need some extension to the language.

For the output, we need no extension and we can express the output of a $1\bot$-sequence with two heads in a functional language.

The output from the first head is expressed as `c:foo()` with `c` the character 0 or 1 and `foo()` the recursive call to produce the rest of the output. The output from the second head is written as `x:c:xs where x:xs=foo()`, with the same meanings for `c` and `foo()`. Note that the new head positions (i.e. `x` and the head of `xs`) comes to be the first two positions of the output of the recursive call of `foo()`. Therefore, we can consider that the tape is composed only of unfilled cells and the two heads are always located at the first two cells of the output. Note that `c` is a constant and therefore, when a term denoting a $1\bot$-stream is reduced to a weak head normal form (i.e., a cons cell), it must have one of the four forms $0 : M'$, $1 : M'$, $x : 0 : M'$, and $x : 1 : M'$.

As for the recursive call, the recursive function may have additional arguments to convey the internal state of the computation. However, in some cases, we can simplify (or even omit) such arguments by allowing the function to modify the result of the recursive call. As such a modification on $1\bot$-sequences, we consider inversion of boolean values. We use the function nh to invert the first character of a string defined as follows:

```
not 0 = 1
not 1 = 0
nh c:a = not c:a
```

and allow expressions like `c:nh foo()` and `not x:c:nh xs where x:xs=foo()` for the programs in the above paragraph.

As an example, we consider the function *stog* which converts the signed-digit representation to the Gray-code representation. The signed-digit representation is an expansion of $[-1, 1]$ as an infinite sequence of $\Gamma = \{0, 1, -1\}$, defined as

$$\delta_{sn}(a_1 a_2 \ldots) = \Sigma_{i=1}^{\infty} \{a_i 2^{-i}\}.$$

It is equal to the ordinary binary expansion if we do not use -1, and highly redundant in that every real number has infinitely many representations. For our purpose of writing the conversion with Gray-code, we fix the first character

as 1 and discard it from the sequence so that every sequence denotes a real number in $\mathcal{I}$. We also change the definition of the Gray-code representation so that when $G(x)$ contains a bottom, then the two sequences obtained by filling the bottom cell with 0 and 1 are also representing x.

When the first digit of a signed-digit representation is $-1, 1$ and 0, it means that the number is in the intervals $[0, 1/2]$, $[1/2, 1]$, and $[1/4, 3/4]$, respectively. Note that these three intervals are expressed in Gray-code representation as the output of 0 and 1 from the first head, and the output of 1 from the second head, respectively. We can write in Haskell the conversion from signed-digit to Gray-code representations as follows considering the recursive structures of both representations [16].

```
stog(1:xs) = 1:nh(stog xs)
stog(-1:xs) = 0:stog xs
stog(0:xs) = c:1:nh ds where c:ds= stog xs
```

When we execute `stog([0,0..])`, it will have no output on the display because the result is `[⊥,1,0,0..]`, but when we execute `tail(stog([0,0..]))`, it will produce `[1,0,0,0...` infinitely.

2.4 IM2-Machine Inputs with the Amb Operator

As for the input, we can express it if we can use McCarthy's "amb" operator[11], as follows. In this paper, we consider a variant of the amb operator of type

```
amb:: a -> a -> Amb a
```

where the datatype `Amb a` is defined as `data Amb a = Left a | Right a`. The term `amb M N` is reduced to `Left` V if M has the normal form V, `Right` V' if N has the normal form V', and its computation does not terminate only when both M and N do not have normal forms. When both M and N have normal forms, we have two possibilities and thus it is a nondeterministic operator.

Since an IM2-machine waits for one of the two cells to be filled, we can express it with the `amb` operator of type `Bool -> Bool -> Amb Bool` as follows.

```
foo(a:b:xs) = case (amb a b) of
Left 0 -> ... foo(b:xs) ...
Left 1 -> ... foo(b:xs) ...
Right 0 -> ... foo(a:xs) ...
Right 1 -> ... foo(a:xs) ... .
```

Note again that the argument of the recursive call is an infinite list without the cells we have read in. Also for this case, we allow the modification of the argument of the recursive calls with `nh` and `not` such as `foo (not b:nh xs)`.

As an example, we consider the *gtos* function which converts the Gray-code representation to the signed-digit representation. This program is also constructed from the recursive structures of both representations.

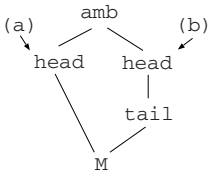


Fig. 3. The sharing structure of the term-graph of `amb`

```
gtos(a:b:xs) = case (amb a b) of
Left 0 -> -1:(gtos (b:xs))
Left 1 -> 1:(gtos (nh (b:xs)))
Right 1 -> 0:(gtos (a:nh xs))
Right 0 -> case a of 0 -> -1: -1:(gtos xs)
                          1 -> 1:1:(gtos (nh xs))
```

3 Partial Sequential Realization of the Amb Operator

As we showed in the previous section, we can express the two-head stream access of an IM2-machine with the amb operator. However, in order to implement this operator, we need to execute two threads for both arguments in parallel. Parallel execution is a heavy mechanism, which is not easy to implement.

When we are implementing the two-head stream access of an IM2-machine, we always use the amb operator in the form

`amb a b where a:b:x =` M`.`

Here, M represents a "producer process" which makes the two-head output access to an $1\bot$-stream. The point is that the calculation of the two arguments of `amb` share the same redex M and therefore we do not need to compute them in parallel. If a functional language is implemented based on graph reduction [7], the above term is represented as shown in Figure 3. Here, an application node is labelled with a combinator name when it is an application of a combinator, for simplicity. In this way, the two term-graphs representing the two arguments share the same subgraph as a redex. Therefore, if we can reduce this term using this sharing structure, it is expected that we can implement the partial amb operator we need for $1\bot$-stream access in the realm of sequential computation. As such an operator, we introduce `gamb`, which has the type

```
gamb:: Bool -> Bool -> Amb Bool.
```

We explain how `gamb` works with an example of the reduction of `gtos(stog [0,0..])`, where the program `gtos` is modified so that it uses the `gamb` operator instead of `amb`. From the definition of `gtos`, the evaluation of `gamb a b` in the definition of `gtos` will produce the term-graph Figure 4(A). The arguments `a b` of `gtos` are marked with (a) and (b), respectively. In this graph, there is only

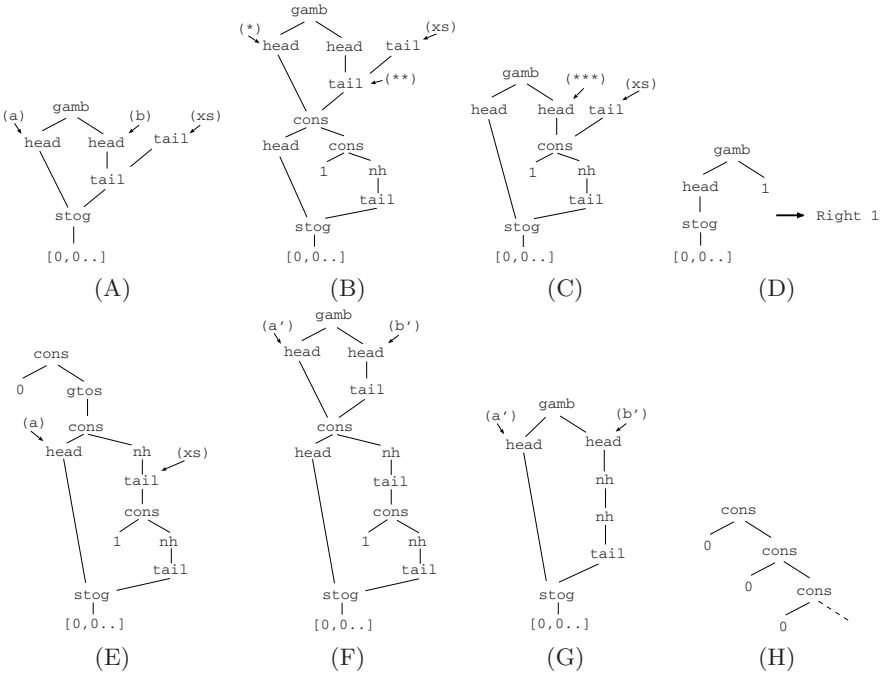


Fig. 4. The evaluation of `gtos(stog([0,0..]))`

one redex node `stog`, which is shared by both of the arguments. Therefore, it is evaluated and we have the term-graph (B). It has three redexes, two of them are put the marks (*) and (**) and the other one is stog. If we use the leftmost outermost reduction strategy as usual functional languages do, the redex (*) is reduced and then the redex `stog` is reduced. After that, it starts a non-terminating computation of the node (a).

However, since the term-graph `stog` is the producer process of an $1\bot$-stream, it is reduced to one of the four forms $0 : M'$, $1 : M'$, $x : 0 : M'$, and $x : 1 : M'$ as we noted in Section 2.3. Therefore, we can obtain one of the normal forms of (a) and (b) by applying only the reduction rules `head` $(B : L) \to B$ and `tail` $(B : L) \to L$. Thus, we consider the reduction strategy of `gamb` MN to reduce both M and N with these two rules as long as they are applicable. Therefore, the redex (**) is reduced before `stog`, and we obtain the term-graph (C). Then, the redex (***) is reduced and we have (D). In this way, the argument (b) is reduced to a normal form 1 and `gamb` will return the value `Right` 1. As the result, the whole program `gtos(stog [0,0..])` is reduced to the term-graph Figure 4(E), which is a head normal form.

Next, consider the reduction of `gtos` in (E), which is a bit more complicated. Note that subgraphs of (xs) are reduced through the reduction of (**) in (B) because of the sharing, and we can use this simplified graph from the beginning. As we noted in Section 2.3 and 2.4, a function to output (input) a $1\bot$-stream

can modify the result (arguments) of the recursive call by inserting `not` and `nh`. Correspondingly, the argument-subgraphs (a') and (b') of the term-graph (F) have nodes with labels `not` and `nh`, above their shared redex `stog`. Thus, we modify the reduction strategy of `amb` MN we mentioned above, so that M and N are reduced with the following reduction rules (I).

Reduction rules (I)

$$\begin{array}{lll}
\texttt{head}\ (B : L) & \to B & \text{(R-head)} \\
\texttt{tail}\ (B : L) & \to L & \text{(R-tail)} \\
\texttt{not}\ 0 & \to 1 & \text{(R-not0)} \\
\texttt{not}\ 1 & \to 0 & \text{(R-not1)} \\
\texttt{nh}\ (B : L) & \to \texttt{not}\ B : L & \text{(R-nh)}
\end{array}$$

This is the minimum list of rules which works. We had better also use the rules `nh nh` $L \to L$ and `not not` $B \to B$ in practice so that the term-graphs do not become significantly large.

Thus, the term-graph (F) is reduced to (G), and then, the two arguments of gamb share the same subgraph `stog`, and thus it is reduced and after that, reduction rules (I) become applicable, and thus it is also reduced to `Right 1`. In this way, if we continue the reduction, we obtain an infinite graph (H).

To summarize, the term-graph `gamb` M N is reduced as follows.

(1) Rules in (I) are applied to subgraphs of M and N which are reachable from the root through nodes labelled with `cons`, `head`, `tail`, `not`, and `nh` until no more rules are applicable. Rules are applied in some fixed order; in our implementation, the leftmost outermost reduction order on M, and then on N.
(2) If one of them become a normal form (that is, 0 or 1), then `gamb` returns the corresponding value. For example, if M is reduced to 0, `gamb` M N is reduced to `Left` 0. If both are normal forms, `gamb` returns the left value.
(3) Compare the leftmost outermost redexes of M and N. If they are not identical, then raise a runtime error.
(4) Reduce the shared redex in (3) to a weak head normal form.
(5) Repeat (1) to (4) until it returns in (2) or it raises an error in (3).

We need the repetition (5) because the shared redex can be reduced to a shared redex again by rules in (I). Rules in (I) are graph reductions implemented as follows. When (R-not0) or (R-not1) is applied, the `not` node is simply overwritten with 1 or 0. It is also the case for the (R-nh) rule; the `nh` node is overwritten with a new `not` node. However, in (R-head) and (R-tail) rules, we cannot overwrite the `head` node with the node B because the node B already exists. Instead, we overwrite `head` with a new indirection node which points to B. This is actually the way term-graphs are reduced in graph-based functional language implementations [9, 7, 6]. We omit an indirection node in the figures.

The `gamb` operator is a partial realization of `amb`. That is, (1) when `gamb` M N is reduced to L, `amb` M N can also be reduced to L, and (2) when the reduction of `gamb` M N does not terminate, the reduction of `amb` M N does not terminate.

4 A Term-Graph Calculus of 1⊥-Streams

In the previous section, we defined the reduction of `gamb` MN for arbitrary terms M and N of type **Bool**, and therefore `gamb` MN may cause a runtime error in (3) depending on the ways graph implementations of M and N share a subgraph. In this section, we define a term-graph calculus $\mathrm{GPCF}^{\omega}_{\perp,1}$ which has the type of 1⊥-streams as a subtype of `[Bool]`, and show that such a runtime error does not happen when `gamb` is used for 1⊥-stream access.

We start with defining a term-graph. Let Γ be a set of labels with arity in $\mathbb{N}$. A term-graph over a signature Γ is a quadruple $g = \langle N, symb, args, r\rangle$ such that (1) N is a finite set of nodes, (2) $symb : N \to \Gamma$ is a function which assigns a label to a node, (3) $args : N \to N^*$ is the list of successor nodes such that $length(args(n)) = arity(symb(n))$, (4) $r \in N$ is the root of g, and (5) g is acyclic as a graph. Note that we only consider acyclic term-graphs in this paper.

$\mathrm{GPCF}^{\omega}_{\perp}$ is a PCF-like term-graph calculus with **Stream** and **AmbBool** type (Figure 5). We consider the minimal set of types for our explanation and omit the integer type, for example. We consider typed variables to simplify the type system and let X^{τ} be a set of variables of type τ. The set Γ of labels (with arity) is defined as $\{0^{(0)}, 1^{(0)}, \mathtt{head}^{(1)}, \mathtt{tail}^{(1)}, :^{(2)}, \mathtt{not}^{(1)}, \mathtt{nh}^{(1)}, x^{\tau(0)}, \lambda x^{\tau(1)}, @^{(2)}(\text{application}), \mathtt{if\_then\_else\_}^{(3)}, \mu x^{\tau(1)}, \mathtt{gambr}^{(1)}, \mathtt{Left}^{(1)}, \mathtt{Right}^{(1)}, ||^{(2)}(\text{destructor for } \mathbf{AmbBool})\}$. Here, a variable x^{τ} is a member of X^{τ}. We have the condition that for each λx^{τ} and μx^{τ}, there exists at most one bounded variable node with label x^{τ}.

We sometimes omit the type τ when it is obvious from the context. Though `not` and `nh` are λ-expressible, we list them as primitives because we need special treatment. Note that the μ constructor does not produce a cyclic graph, and μx^{τ} is a label of a node for each x^{τ}.

When we express a term-graph in text, we use infix notation for :, ||, and @, and we omit the operator symbol @. We consider that multiple occurrences of the same bounded variable are expressing the same variable node.

In order to express sharing of nodes, we assign a variable to each node which has more than one parents, and use the notation M `where` $x^{\tau} = N$ for the graph M with the variable node x^{τ} substituted for N of type τ. We also use a notation with pattern matching for the **Stream** type; we write M `where` $y_1 : \ldots : y_n : x = N$ for M `where` $x =$ `tail` z_n `where` $y_n =$ `head` z_n $\ldots$ `where` $z_2 =$ `tail` z_1`where` $y_1 =$ `head` z_1`where` $z_1 = N$. It is close to the call-by-need calculus in [1], but they consider a term-calculus which simulates graph-based reduction, whereas the objects of the calculus are term-graphs themselves in our calculus. We also write `not`^{n} M for the n-times successive application of `not` to M.

In this type system, all the graphs are directed acyclic graphs, and therefore the type of a term-graph is uniquely defined inductively. Reduction rules are graph-reduction rules. Note that M `where` $x^{\tau} = N$, which is the right hand side of (R-app) is obtained by copying M so that edges pointing to x^{τ} are redirected to N. It is also the case for the (R-mu) rule. The set of reduction rules is composed of the (I)-reduction and rules in (II) and (III). The reduction

$\mathbf{GPCF}^{\omega}_{\perp,1}$

Types: $\sigma, \tau ::= \mathbf{Bool} \mid \mathbf{Stream} \mid \sigma \to \tau \mid \mathbf{AmbBool}$
Variables(of type τ): x^{τ}, y^{τ}, z^{τ}
Term-Graphs: $B, L, M, N ::= 0 \mid 1 \mid \mathtt{head}\ L \mid \mathtt{tail}\ L \mid B : L \mid \mathtt{not}\ B \mid \mathtt{nh}\ B$
$\mid \lambda x^{\tau}.M \mid M\ N \mid \mu x^{\tau}.M \mid \mathtt{if}\ B\ \mathtt{then}\ M\ \mathtt{else}\ N$
$\mid \mathtt{gambr}\ L \mid \mathtt{Left}\ B \mid \mathtt{Right}\ B \mid M||N$

Typesystem :

$$x^{\sigma} :: \sigma \qquad 0 :: \mathbf{Bool} \qquad 1 :: \mathbf{Bool} \qquad \frac{B :: \mathbf{Bool}}{\mathtt{not}\ B :: \mathbf{Bool}}$$

$$\frac{L :: \mathbf{Stream}}{\mathtt{head}\ L :: \mathbf{Bool}} \qquad \frac{L :: \mathbf{Stream}}{\mathtt{tail}\ L :: \mathbf{Stream}} \qquad \frac{L :: \mathbf{Stream}}{\mathtt{nh}\ L :: \mathbf{Stream}}$$

$$\frac{M :: \tau}{\lambda x^{\sigma}.M :: \sigma \to \tau} \qquad \frac{B :: \mathbf{Bool}, M :: \sigma, N :: \sigma}{\mathtt{if}\ B\ \mathtt{then}\ M\ \mathtt{else}\ N :: \sigma} \qquad \frac{M :: \sigma \to \tau, N :: \sigma}{MN :: \tau}\ \text{(T-tail)}$$

$$\frac{M :: \sigma}{\mu x^{\sigma}.M :: \sigma}\ \text{(T-mu)} \qquad \frac{L :: \mathbf{Stream}}{\mathtt{not}^n\ c : L\ ::\ \mathbf{Stream}}\ (n \geq 0, c = 0 \text{ or } c = 1)\ \text{(T-cons)}$$

$$\frac{L :: \mathbf{Stream}}{\mathtt{not}^l\ y : c_1 : \ldots : c_m : \mathtt{nh}^n\ x\ \ \mathtt{where}\ y : z_1 : \ldots : z_k : x = L\ ::\ \mathbf{Stream}}$$
$$(c_i = 0 \text{ or } c_i = 1 (i = 1, \ldots, m), l, m, n, k \geq 0)\quad \text{(T-b01)}$$

$$\frac{L :: \mathbf{Stream}}{\mathtt{gambr}\ L :: \mathbf{AmbBool}} \qquad \frac{L :: \mathbf{AmbBool},\quad M :: \mathbf{Bool} \to \sigma,\quad N :: \mathbf{Bool} \to \sigma}{(M||N)\ L :: \sigma}$$

$$\frac{B :: \mathbf{Bool}}{\mathtt{Left}\ B :: \mathbf{AmbBool}} \qquad \frac{B :: \mathbf{Bool}}{\mathtt{Right}\ B :: \mathbf{AmbBool}}$$

(I)-reduction: applying the following rules to subgraphs reachable from the root through nodes labelled with :, `head`, `tail`, `not`, and `nh` until no more rules are applicable.

$$\begin{array}{lll}
\mathtt{head}\ (B : L) \to B & & \text{(R-head)} \\
\mathtt{tail}\ (B : L) \to L & & \text{(R-tail)} \\
\mathtt{not}\ 0 \to 1 & & \text{(R-not0)} \\
\mathtt{not}\ 1 \to 0 & & \text{(R-not1)} \\
\mathtt{nh}\ (B : L) & \to \mathtt{not}\ B : L & \text{(R-nh)}
\end{array}$$

Reduction rules (II):

$$\begin{array}{lll}
(\lambda x^{\tau}.M)\ N & \to M\ \mathtt{where}\ x^{\tau} = N & \text{(R-app)} \\
\mu x^{\tau}.M & \to M\ \mathtt{where}\ x^{\tau} = \mu x^{\tau}.M & \text{(R-mu)} \\
\mathtt{if}\ 1\ \mathtt{then}\ M\ \mathtt{else}\ N & \to M & \text{(R-if0)} \\
\mathtt{if}\ 0\ \mathtt{then}\ M\ \mathtt{else}\ N & \to N & \text{(R-if1)} \\
f||g\ (\mathtt{Left}\ B) & \to f\ B & \text{(R-L)} \\
f||g\ (\mathtt{Right}\ B) & \to g\ B & \text{(R-R)}
\end{array}$$

Reduction rules (III):

$$\begin{array}{ll}
\mathtt{gambr}\ (0 : M) \to \mathtt{Left}\ 0 & \mathtt{gambr}\ (M : 0 : N) \to \mathtt{Right}\ 0 \\
\mathtt{gambr}\ (1 : M) \to \mathtt{Left}\ 1 & \mathtt{gambr}\ (M : 1 : N) \to \mathtt{Right}\ 1
\end{array}$$

Fig. 5. The term-graph calculus of 1⊥-streams $\mathrm{GPCF}^{\omega}_{\perp,1}$

rules are parallel to the evaluation rules of `gamb` in Section 3. We have defined the (I)-reduction in this form for two reasons. One is to provide the subject-reduction property, and the other one is to reduce B_2 to a (I)-normal form when `gambr`$(B_1 : B_2 : L)$ is given. We consider leftmost outermost reduction order.

As operations to construct **Stream** term-graphs, we consider insertion of a constant to the head or next to the head of a **Stream** element. This is parallel to the constructor of ordinary streams to insert a constant to the head. We also allow the operation to remove the first or the second element from a **Stream** element to form a new $1\bot$-stream. The (T-cons) and (T-tail) rules are for the operations to the first element, and (T-b01) rule with $m = 1, k = 0$ (or $m = 0, k = 1$) is for the insertion (or removal, respectively) operation to the second element. The (T-b01) rule is applicable to terms-graphs obtained through successive applications of these operators. As for destructors, we use the `gambr` operator which corresponds to the following program.

```
gambr(M) = gamb a b where a:b:x = M.
```

Note that this is the way `gamb` is used to input from a $1\bot$-stream. By structural induction on M, we can prove the following.

Proposition 1. *Suppose that $M :: \sigma$ and M is reduced to N in GPCF$^{\omega}_{\bot,1}$. Then N is typable and $N :: \sigma$.*

Corollary 1. *Suppose that M is a term of type* **Stream** *and M is reduced to $L = N_1 : N_2$. Then, after applying (I)-reduction, one of the followings hold,*
(1) N_1 is a constant (i.e. 0 or 1),
(2) N_2 is $c : N_3$ for c a constant,
(3) L has the form $\texttt{not}^l\ y : \texttt{nh}^n\ x$ `where` $y : z_1 : \ldots : z_m : x = G$ *for $l, m, n \geq 0$.*

Corollary 1 shows that reduction of `gambr` M with the leftmost outermost reduction will produce a value (case (1) or (2)), or continue the reduction of G. For the latter case, when G is reduced to a cons cell, L is reduced by (I)-reduction to one of the forms (1), (2), and (3). Thus, the reduction of `gambr` M will produce a value `Left` c or `Right` c ($c = 0$ or 1) or it does not terminate.

Thus, we can say that runtime error does not occur for a typable program when `gamb` is added to a graph-reduction based lazy functional language with this kind of graph-representations. However, most implementations of Haskell use cyclic graphs for the representation of recursive structures. For this case, typing rules presented here do not have such good properties. We consider a variant GPCF$^{\omega,c}_{\bot,1}$ of GPCF$^{\omega}_{\bot,1}$ in which term-graphs are allowed to be cyclic, $\mu x^{\tau}.M$ is not a term-graph but a textual representation of a cyclic graph, and (T-mu) and (R-mu) do not exit. To see the difference, consider the term-graph K `=`μ`x. a:1:y where a:y = x` of type **Stream**. In GPCF$^{\omega}_{\bot,1}$, it is reduced to **Stream**-type terms of the form `a:1:1:1:...:1:y where a:y =` K`.` On the other hand, in GPCF$^{\omega,c}_{\bot,1}$, it is reduced by (I)-reduction to the term-graph `(`μ`x.x):(`μ`y.1:y)` which does not belong to **Stream** type. Here, μ`x.x` is an indirection node pointing itself. Roughly speaking, GPCF$^{\omega}_{\bot,1}$ step by step simulates two-head stream

output of an IM2-machine, whereas cyclic graph representation enables us to make all the infinite outputs at a time and realizes the result of infinite-time computation, which our type system does not deal with. In the same way, the reduction of the term-graph `gambr(`μ`x. a:y where a:y = x)` of type **AmbBool** does not terminate in $\mathrm{GPCF}^{\omega}_{\perp,1}$ whereas it is reduced to `gambr((`μ`a.a):(`μ`y.y))` by (I)-reduction and stops (i.e., causing a runtime error) in $\mathrm{GPCF}^{\omega,c}_{\perp,1}$. Our implementation in the next section is close to $\mathrm{GPCF}^{\omega,c}_{\perp,1}$ and has this behavior.

5 Implementation

We have implemented our `gamb` operator as an extension of the Hugs system, which is a graph-reduction based implementation of the Haskell language. First, we implemented it as an extension of Gofer ver2.30 which is an ancestor of the Hugs system. Because Gofer ver2.30 has a good documentation [6], in particular of the G-machine structure of the runtime system [7], it is not difficult to put a hook on the `eval` operator of the G-machine of the Gofer system so that when it is a function application and the function is `gamb`, then reduce it as listed in Section 3. This implementation is available from the author's web page[18].

6 Some Algorithms with Gamb

As we explained in the introduction, the main application area of $1\perp$-stream programming is real number computation. We list two programs to compute real-functions over the unit interval [0,1] in [18]. One is the average function `pl` to compute $(x + y)/2$, and the other one is multiplication. They can also be expressed as $\mathrm{GPCF}^{\omega}_{\perp,1}$-terms of type **Stream** $\rightarrow$ **Stream** $\rightarrow$ **Stream**.

We can also apply higher-order functions "map" and "foldr1" to real functions like `pl`. Therefore, for example, we can write and execute

```
sum a = gtos (foldr1 pl (map stog (map (code a))))
```

which calculates the sum of the elements of the finite list `a`. Here, the `code` function maps a real number with decimal representation to the binary representation. This time again, it can also be expressed in $\mathrm{GPCF}^{\omega}_{\perp,1}$.

7 Conclusion

We extended the notion of a stream to a stream with at most one bottom and implemented, as an extension of Haskell, input/output of such extended streams. This mechanism can be used for real number computation because the set of real numbers is topologically embedded in $\Sigma^{\omega}_{\perp,1}$.

We defined a datatype corresponding to a $1\perp$-stream as a subtype of the infinite list type [`Bool`]. There is another way of implementing computation over $1\perp$-streams. That is, to assign a name to each constructor and represent a $1\perp$-stream as an ordinary stream. However, because of the several ways of

constructing the same $1\perp$-stream, such representation is not canonical. From the authors experience, the existence of multiple-representation complicates $1\perp$-stream programs. In addition, if we accept this approach, we need to consider the relation between the denotation as a $1\perp$-stream and its representation as an ordinary stream. Since a $1\perp$-stream itself is directly expressible in a programming language, the author thinks it natural to try to write a program which directly manipulates them, as we did in this paper.

We have two goals in this study of $1\perp$-stream calculi. One is to actually implement it and write and execute real-number programs. The other one is to study the computational structure of $1\perp$-streams and relate it to that of real numbers. One observation here is that, the nondeterminism and multi-valuedness of functions over $1\perp$-streams appear not because we perform parallel computation but because we access the *intensional* information how the arguments are represented as term-graphs. We need to investigate it with non-sequentiality feature of real number computation studied in [3].

In this paper, we have only presented the operational side of $\mathrm{GPCF}^{\omega}_{\perp,1}$. It is expected that, through the investigation of the denotational side of $\mathrm{GPCF}^{\omega}_{\perp,1}$, we can study the structure of $1\perp$-streams from many aspects, including algebraic, domain-theoretic, and category-theoretic point of view. The author is interested in applying the semantics of a sequential nondeterministic language in [10] to our language. It is left as a future work.

Acknowledgments. The authors thanks Martin Escardo for discussions about the use of amb operator for Gray-code computation. The first author was supported in part by Kayamori Foundation of Informatical Science Advancement.

References

1. Zena Ariola, Matthias Felleisen, John Maraist, Martin Odersky and Philip Wadler. A Call-by-Need Lambda Calculus, in *Proc. POPL '95, 22'nd Annual Symposium on Principles of Programming Languages, San Francisco, California*, 233-246, 1995.
2. A. Du Bois, R. Pointon, H.-W. Loidl, and P. Trinder. Implementing Declarative Parallel Bottom-Avoiding Choice. in *Proc. 14th Symposium on Computer Architecture and High Performance Computing*, 2002.
3. Martín Hötzel Escardó, Martin Hofmann, and Thomas Streicher. On the non-sequential nature of the interval-domain model of exact real-number computation. *Mathematical Structures in Computer Science*, to appear.
4. Pietro Di Gianantonio. An Abstract Data Type for Real Numbers. *Theoretical Computer Science*, 221:295–326, 1999.
5. F. Gray. Pulse code communications. U. S. Patent 2632058, March 1953.
6. Mark P. Jones. The implementation of the Gofer functional programming system. *Research Report YALEU/DCS/RR-1030*, Yale University, USA, 1994.
7. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
8. Simon Peyton Jones, Editor. *Haskell 98 Language and Libraries, The Revised Report.* Cambridge University Press, 2003.
9. J.R. Kennaway, J.W. Klop, M.R. Sleep and F.J. de Vries. An Introduction to Term Graph Rewriting. in *[14]*

10. J. Raymundo Marcial-Romero and Martín Hötzel Escardó. Semantics of a Sequential Language for Exact Real-Number Computation. in *Proceedings of the Annual IEEE Symbposium on Logic in Computer Science*. 426–435, 2004.
11. John McCarthy. A Basis for a Mathematical Theory of Computation. in P. Braffort and D. Hirshberg, editors, *Computer Programming and Formal Systems*, 33–70, North-Holland. 1963.
12. John Hughes and Andrew Moran. Making Choices Lazily. in *Conference Record of FPCA '95*, 108–119,ACM Press, 1995.
13. Kristoffer H. Rose Graph-based Operational Semantics of a Lazy Functional Language. in *[14]*
14. Ronan Sleep, Rinus Plasmeijer and Marko van Eekelen, Editors. *Term Graph Rewriting: Theory and Practice*. John Wiley & Sons Ltd, 1993.
15. Hideki Tsuiki. Computational dimension of topological spaces. In *Computability and Complexity in Analysis*, LNCS 2064, 323–335, Springer,2001.
16. Hideki Tsuiki. Real number computation through gray code embedding. *Theoretical Computer Science*, 284(2):467–485, 2002.
17. Hideki Tsuiki. Real Number Computation with Committed Choice Logic Programming. *Journal of Logic and Algebraic Programming*, to appear, 2004.
18. `http://www.i.h.kyoto-u.ac.jp/~tsuiki/bot_haskell`
19. Klaus Weihrauch. *Computable analysis, an Introduction*. Springer-Verlag, 2000.

Bottom-Up β-Reduction: Uplinks and λ-DAGs

Olin Shivers[1] and Mitchell Wand[2]

[1] Georgia Institute of Technology
[2] Northeastern University

Abstract. Representing a λ-calculus term as a DAG rather than a tree allows us to represent the sharing that arises from β-reduction, thus avoiding combinatorial explosion in space. By adding uplinks from a child to its parents, we can efficiently implement β-reduction in a bottom-up manner, thus avoiding combinatorial explosion in time required to search the term in a top-down fashion. We present an algorithm for performing β-reduction on λ-terms represented as uplinked DAGs; discuss its relation to alternate techniques such as Lamping graphs, explicit-substitution calculi and director strings; and present some timings of an implementation. Besides being both fast and parsimonious of space, the algorithm is particularly suited to applications such as compilers, theorem provers, and type-manipulation systems that may need to examine terms in-between reductions—*i.e.*, the "readback" problem for our representation is trivial. Like Lamping graphs, and unlike director strings or the suspension λ-calculus, the algorithm functions by side-effecting the term containing the redex; the representation is *not* a "persistent" one. The algorithm additionally has the charm of being quite simple: a complete implementation of the core data structures and algorithms is 180 lines of SML.

1 Introduction

The λ-calculus [2, 5] is a simple language with far-reaching use in the programming-languages and formal-methods communities, where it is frequently employed to represent, among other objects, functional programs, formal proofs, and types drawn from sophisticated type systems. Here, our particular interest is in the needs of client applications such as compilers, which may use λ-terms to represent both program terms as well as complex types. We are somewhat less focussed on the needs of graph-reduction engines, where there is greater representational license—a graph reducer can represent a particular λ-term as a chunk of machine code (*e.g.*, by means of supercombinator extraction), because its sole focus is on *executing* the term. A compiler, in contrast, needs to examine, analyse and transform the term in-between reductions, which requires the actual syntactic form of the term be available at the intermediate steps.

Of the three basic operations on terms in the λ-calculus—α-conversion, β-reduction, and η-reduction—it is β-reduction that accomplishes the "heavy lifting" of term manipulation. (The other two operations are simple to implement.) Unfortunately, naïve implementations of β-reduction can lead to exponential time and space blowup.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 217–232, 2005.

There are only three forms in the basic language: λ expressions, variable references, and applications of a function to an argument:

$$t \in \text{Term} ::= \lambda x.t \mid x \mid t_f\ t_a$$

where "x" stands for a member of some infinite set of variables.

β-reduction is the operation of taking an application term whose function subterm is a λ-expression, and substituting the argument term for occurrences of the λ's bound variable in the function body. The result, called the *contractum,* can be used in place of the original application, called the *redex*. We write

$$(\lambda x.b)\, a \quad \Rightarrow \quad [x \mapsto a]b$$

to express the idea that the redex applying function $\lambda x.b$ to argument a reduces to the contractum $[x \mapsto a]b$, by which we mean term b, with free occurrences of x replaced with term a.

We can define the core substitution function with a simple recursion:

$$\begin{array}{lll}
[y \mapsto t][\![x]\!] & = t & x = y \\
[y \mapsto t][\![x]\!] & = x & x \neq y \\
[x \mapsto t][\![t_f\ t_a]\!] & = ([x \mapsto t]t_f)([x \mapsto t]t_a) & \\
[x \mapsto t][\![\lambda y.b]\!] & = \lambda y'.([x \mapsto t][y \mapsto y']b) & y' \text{ fresh in } b \text{ and } t.
\end{array}$$

Note that, in the final case above, when we substitute a term t under a λ-expression $\lambda y.b$, we must first replace the λ-expression's variable y with a fresh, unused variable y' to ensure that any occurrence of y in t isn't "captured" by the $[x \mapsto t]$ substitution. If we know that there are no free occurrences of y in t, this step is unnecessary—which is the case if we adopt the convention that every λ-expression binds a unique variable.

It is a straightforward matter to translate the recursive substitution function defined above into a recursive procedure. Consider the case of performing a substitution $[y \mapsto t]$ on an application $t_f\ \ t_a$. Our procedure will recurse on both subterms of the application... but we could also use a less positive term in place of "recurse" to indicate the trouble with the algorithmic handling of this case: search. In the case of an application, the procedure will blindly search *both* subterms, even though one or both may have no occurrences of the variable for which we search. Suppose, for example, that the function subterm t_f, is very large—perhaps millions of nodes—but contains no occurrences of the substituted variable y. The recursive substitution will needlessly search out the entire subterm, constructing an identical copy of t_f. What we want is some way to direct our recursion so that we don't waste time searching into subterms that do not contain occurrences of the variable being replaced.

2 Guided Tree Substitution

Let's turn to a simpler task to develop some intuition. Consider inserting an integer into a set kept as an ordered binary tree (Fig. 1). There are three things about this simple algorithm worth noting:

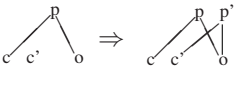
```
Procedure addItem(node, i)
  if node = nil then
      new := NewNode()
      new.val   := i
      new.left  := nil
      new.right := nil
  else if node.val < i then
      new := NewNode()
      new.val   := node.val
      new.left  := node.left
      new.right := addItem(node.right,i)
  else if node.val > i then
      new := NewNode()
      new.val   := node.val
      new.right := node.right
      new.left  := addItem(node.left,i)
  else new := node
  return new
```

Fig. 1. Make a copy of ordered binary tree *node*, with added entry *i*. The original tree is not altered

- **No search**
  The pleasant property of ordered binary trees is that we have enough information as we recurse down into the tree to proceed only into subtrees that require copying.
- **Steer down; build up**
  The algorithm's recursive control structure splits decision-making and the actual work of tree construction: the downward recursion makes the decisions about which nodes need to be copied, and the upward return path assembles the new tree.
- **Shared structure**
  We copy only nodes along the spine leading from targeted node to the root; the result tree shares as much structure as possible with the original tree.

3 Guiding Tree Search with Uplinks

Unfortunately, in the case of β-reduction, there's no simple, compact way of determining, as we recurse downwards into a tree, which way to go at application nodes—an application has two children, and we might need to recurse into one, the other, both, or neither. Suppose, however, that we represent our tree using not only down-links that allow us to go from a parent to its children, but also with redundant up-links that allow us to go from a child to its parent. If we can (easily) find the leaf node in the original tree we wish to replace, we can chase uplinks along the spine from the old leaf to the tree root, copying as we go (Fig. 2). The core iteration of this algorithm is the $c \mapsto c'$ upcopy:



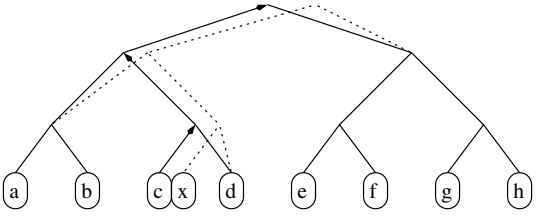


Fig. 2. Replacing a single leaf in a binary tree by following uplinks. Here, we make a copy of the original tree, replacing leaf c with x

We take a child c and its intended replacement c', and replicate the parent p of c, making the $c \mapsto c'$ substitution. This produces freshly-created node p'; we can now iterate, doing a $p \mapsto p'$ upcopy into the parent of p at the next step, and so on, moving up through the original tree until we reach the root.

Note the similar properties this upcopy algorithm has with the previous algorithm: no search required; we build as we move upwards; we share as much structure as possible with the old tree, copying only the nodes along the "spine" leading from the leaf back to the root. For a balanced tree, the amount of copying is logarithmic in the total number of nodes. If we can somehow get our hands on the leaf node to be replaced in the old tree, the construction phase just follows uplinks to the root, instead of using a path saved in the recursion stack by the downwards search.

4 Upcopy with DAGs

We can avoid space blowup when performing β-reduction on λ-calculus terms if we can represent them as directed acyclic graphs (DAGs), not trees. Allowing sharing means that when we substitute a large term for a variable that has five or six references inside its binding λ-expression, we don't have to create five or six distinct copies of the term (that is, one for each place it occurs in the result). We can just have five or six references to the same term. This has the potential to provide logarithmic compression on the simple representation of λ-calculus terms as trees. These term DAGs can be thought of as essentially a space-saving way to represent term trees, so we can require them, like trees, to have a single top or root node, from which all other nodes can be reached.

When we shift from trees to DAGs, however, our simple functional upcopy algorithm no longer suffices: we have to deal with the fact that there may be multiple paths from a leaf node (a variable reference) of our DAG up to the root of the DAG. That is, any term can have multiple parents. However, we can modify our upwards-copying algorithm in the standard way one operates on DAGs: we search upwards along all possible paths, marking nodes as we encounter them. The first time we copy up into a node n, we replicate it, as in the previous tree algorithm, and continue propagating the copy operation up the tree to the (possibly multiple) parents of n. However, before we move upwards from n, we first store the copy n' away in a "cache" field of n. If we later copy up into n

```
Procedure upcopy(childcopy, parent, relation)
  if parent.cache is empty then
      parcopy := NewNode()
      if relation is "left child" then
      parcopy.left  := childcopy
      parcopy.right := parent.right
      else
      parcopy.right := childcopy
      parcopy.left  := parent.left
      parent.cache := parcopy
      for-each <grandp,gprel> in parent.uplinks do
    upcopy(parcopy, grandp, gprel)
  else
      parcopy := parent.cache
      if relation is "left child"
      then parcopy.left  := childcopy
      else parcopy.right := childcopy
```

Fig. 3. Procedure upcopy makes a copy of a binary DAG, replacing the *relation* child (left or right) of *parent* with *childcopy*

via its other child, the presence of the copy n' in the cache slot of n will signal the algorithm that it should not make a second copy of n, and should not proceed upwards from n—that has already been handled. Instead, it mutates the existing copy, n', and returns immediately.

The code to copy a binary DAG, replacing a single leaf, is shown in Fig. 3. Every node in the DAG maintains a set of its uplinks; each uplink is represented as a $\langle \mathit{parent}, \mathit{relation} \rangle$ pair. For example, if node c is the left child of node p, then the pair $\langle p, \text{left-child} \rangle$ will be one of the elements in c's uplink set.

The upcopy algorithm explores each edge on all the paths from the root of the DAG to the copied leaf exactly once; marking parent nodes by depositing copies in their cache slots prevents the algorithm from redundant exploration. Hence this graph-marking algorithm runs in time proportional to the number of edges, *not* the number of paths (which can be exponential in the number of edges). Were we to "unfold" the DAG into its equivalent tree, we would realise this exponential blowup in the size of the tree, and, consequently, also in the time to operate upon it. Note that, analogously to the tree-copying algorithm, the new DAG shares as much structure as possible with the old DAG, only copying nodes along the spine (in the DAG case, spines) from the copied leaf to the root.

After an upcopy has been performed, we can fetch the result DAG from the cache slot of the original DAG's root. We must then do another upwards search along the same paths to clear out the cache fields of the original nodes that were copied, thus resetting the DAG for future upcopy operations. This cache-clearing pass, again, takes time linear in the number of edges occurring on the paths from the copied leaf to the root. (Alternatively, we can keep counter fields on the nodes to discriminate distinct upcopy operations, and perform a global reset on the term when the current-counter value overflows.)

5 Operating on λ-DAGs

We now have the core idea of our DAG-based β-reduction algorithm in place, and can fill in the details specific to our λ-expression domain.

Basic representation. We will represent a λ-calculus term as a rooted DAG.

Sharing. Sharing will be generally allowed, and sharing will be *required* of variable-reference terms. That is, any given variable will have no more than one node in the DAG representing it. If one variable is referenced by (is the child of) multiple parent nodes in the graph, these nodes will simply all contain pointers to the same data structure.

Bound-variable short-cuts. Every λ-expression node will, in addition to having a reference to its body node, also have a reference to the variable node that it binds. This, of course, is how we navigate directly to the leaf node to replace when we begin the upcopy for a β-reduction operation. Note that this amounts to an α-uniqueness condition—we require that every λ-expression bind a unique variable.

Cache fields. Every application node has a cache field that may either be empty or contain another application node. λ-expression nodes do not need cache fields—they only have one child (the body of the λ-expression), so the upcopy algorithm can only copy up through a λ-expression once during a β-reduction.

Uplinks. Uplinks are represented by $\langle \mathit{parent}, \mathit{relation} \rangle$ pairs, where the three possible relations are "λ body," "application function," and "application argument." For example, if a node n has an uplink $\langle l, \lambda\text{-body} \rangle$, then l is a λ-expression, and n is its body.

Copying λ-expressions. With all the above structure in place, the algorithm takes shape. To perform a β-reduction of redex $(\lambda x.b)\, a$, where b and a are arbitrary subterms, we simply initiate an $x \mapsto a$ upcopy. This will copy up through all the paths connecting top node b and leaf node x, building a copy of the DAG with a in place of x, just as we desire.

Application nodes, having two children, are handled just as binary-tree nodes in the general DAG-copy algorithm discussed earlier: copy, cache & continue on the first visit; mutate the cached copy on a second visit. λ-expression nodes, however, require different treatment. Suppose, while we are in the midst of performing the reduction above, we find ourselves performing a $c \mapsto c'$ upcopy, for some internal node c, into a λ parent of c: $\lambda y.c$. The general structure of the algorithm calls for us to make a copy of the λ-expression, with body c'. But we must also allocate a fresh variable, y', for our new λ-expression, since we require all λ-expressions to bind distinct variables. This gives us $\lambda y'.c'$. Unfortunately, if old body c contains references to y, these will also occur in c'—not y'. We can be sure c' contains no references to y', since y' was created after c'! We need to fix up body c' by replacing all its references to y with references to y'.

Luckily, we already have the mechanism to do this: before progressing upwards to the parents of $\lambda y.c$, we simply initiate a $y \mapsto y'$ upcopy through the existing DAG. This upcopy will proceed along the paths leading from the y reference, up through the DAG,

to the $\lambda y.c$ node. If there are such paths, they *must* terminate on a previously-copied application node, at which point the upcopy algorithm will mutate the cached copy and return.

Why must these paths all terminate on some previously copied application node? Because we have already traversed a path from x up to $\lambda y.c$, copying and caching as we went. Any path upwards from the y reference must eventually encounter $\lambda y.c$, as well—this is guaranteed by lexical scope. The two paths must, then, converge on a common application node—the only nodes that have two children. That node was copied and cached by the original x-to-$\lambda y.c$ traversal.

When the $y \mapsto y'$ upcopy finishes updating the new DAG structure and returns, the algorithm resumes processing the original $c \mapsto c'$ upcopy, whose next step is to proceed upwards with a $(\lambda y.c) \ \mapsto \ (\lambda y'.c')$ upcopy to all of the parents of $\lambda y.c$, secure that the c' sub-DAG is now correct.

The single-DAG requirement. We've glossed over a limitation of the uplink representation, which is that a certain kind of sharing is not allowed: after a β-reduction, the original redex must die. That is, the model we have is that we start with a λ-calculus term, represented as a DAG. We choose a redex node somewhere within this DAG, reduce it, and *alter the original DAG to replace the redex with the contractum*. When done, the original term has been changed—where the redex used to be, we now find the contractum. What we *can't* do is to choose a redex, reduce it, and then continue to refer to the redex or maintain an original, unreduced copy of the DAG. Contracting a redex kills the redex; the term data structure is not "pure functional" or "persistent" in the sense of the old values being unchanged. (We can, however, first "clone" a multiply-referenced redex, splitting the parents between the original and the clone, and then contract only one of the redex nodes.)

This limitation is due to the presence of the uplinks. They mean that a subterm can belong to only one rooted DAG, in much the same way that the backpointers in a doubly-linked list mean that a list element can belong to only one list (unlike a singly-linked list, where multiple lists can share a common tail). The upcopy algorithm assumes that the uplinks exactly mirror the parent→child downlinks, and traces up through all of them. This rules out the possibility of having a node belong to multiple distinct rooted DAGs, such as a "before" and "after" pair related by the β-reduction of some redex occurring within the "before" term.

Hence the algorithm, once it has finished the copying phase, takes the final step of disconnecting the redex from its parents, and replacing it with the contractum. The redex application node is now considered dead, since it has no parents, and can be removed from the parent/uplink sets of its children and deallocated. Should one of its two children thus have its parent set become empty, it, too, can be removed from the parent sets of its children and deallocated, and so forth. Thus we follow our upwards-recursive construction phase with a downwards-recursive deallocation phase.

It's important to stress, again, that this deallocation phase is not optional. A dead node must be removed from the parent sets of its children, lest we subsequently waste time doing an upcopy from a child up into a dead parent during a later reduction.

Termination and the top application. Another detail we've not yet treated is termination of the upcopy phase. One way to handle this is simply to check as we move up through the DAG to see if we've arrived at the λ-expression being reduced, at which point we could save away the new term in some location and return without further upward copying. But there is an alternate way to handle this. Suppose we are contracting redex$(\lambda x.b)\ n$, for arbitrary sub-terms b and n. At the beginning of the reduction operation, we first check to see if x has no references (an easy check: is its uplink set empty?). If so, the answer is b; we are done.

Otherwise, we begin at the λ-expression being reduced and scan downwards from λ-expression to body, until we encounter a non-λ-expression node—either a variable or an application. If we halt at a variable, it *must* be x—otherwise x would have no references, and we've already ruled that out. This case can also be handled easily: we simply scan back through this chain of nested λ-expressions, wrapping fresh λ-expressions around n as we go.

Finally, we arrive at the general case: the downward scan halts at the topmost application node a of sub-term b. We make an identical copy a' of a, *i.e.* one that shares both the function and argument children, and install a' in the cache slot of a.

Now we can initiate an $x \mapsto n$ upcopy, knowing that all upwards copying must terminate on a previously-copied application node. This is guaranteed by the critical, key invariant of the DAG: all paths from a variable reference upward to the root *must* encounter the λ-node binding that variable—this is simply lexical-scoping in the DAG context. The presence of a' in the cache slot of a will prevent upward copying from proceeding above a. Node a acts as a sentinel for the search; we can eliminate the root check from the upcopy code, for time savings.

When the upcopy phase finishes, we pass a' back up through the nested chain of λ-expressions leading from a back to the top $\lambda x.b$ term. As we pass back up through each λ-expression $\lambda y.t$, we allocate a fresh λ-expression term and a fresh variable y' to wrap around the value t' passed up, then perform a $y \mapsto y'$ upcopy to fix up any variable references in the new body, and then pass the freshly-created $\lambda y'.t'$ term on up the chain. (Note that the extended example shown in Sec. 7 omits this technique to simplify the presentation.)

6 Fine Points

These fine points of the algorithm can be skipped on a first reading.

Representing uplinks. A node keeps its uplinks chained together in a doubly-linked list, which allows us to remove an uplink from a node's uplink set in constant time. We will need to do this, for example, when we mutate a previously copied node n to change one of its children—the old child's uplink to n must be removed from its uplink set.

We simplify the allocation of uplinks by observing that each parent node has a fixed number of uplinks pointing to it: two in the case of an application and one in the case of a λ-expression. Therefore, we allocate the uplink nodes along with the parent, and thread the doubly-linked uplink lists through these pre-allocated nodes.

An uplink doubly-linked list element *appears* in the uplink list of the child, but the element *belongs* to the parent. For example, when we allocate a new application node, we simultaneously allocate two uplink items: one for the function-child uplink to the application, and one for the argument-child uplink to the application. These three data structures have identical lifetimes; the uplinks live as long as the parent node they reference. We stash them in fields of the application node for convenient retrieval as needed. When we mutate the application node to change one of its children, we also shift the corresponding uplink structure from the old child's uplink list to the new child's uplink list, thus keeping the uplink pointer information consistent with the downlink pointer information.

The single-reference fast path. Consider a redex $(\lambda x.b)\ n$, where the λ-expression being reduced has exactly one parent. We know what that parent must be: the redex application itself. This application node is about to die, when all references to it in the term DAG are replaced by references to the contractum. So the λ-expression itself is about to become completely parentless—*i.e.*, it, too, is about to die. This means that any node on a path from x up to the λ-expression will also die. Again, this is the key invariant provided by lexical scope: all paths from a variable reference upward to the root *must* encounter the λ-expression binding that variable. So if the λ-expression has no parents, then all paths upwards from its variable must terminate at the λ-expression itself.

This opens up the possibility of an alternate, fast way to produce the contractum: when the λ-expression being reduced has only one parent, mutate the λ-expression's body, altering all of x's parents to refer instead to n. We do no copying at all, and may immediately take the λ-expression's body as our answer, discarding the λ-expression and its variable x (in general, a λ-expression and its variable are always allocated and deallocated together).

Opportunistic iteration. The algorithm can be implemented so that when a node is sequencing through its list of uplinks, performing a recursive upcopy on each one, the final upcopy can be done with a tail recursion (or, if coded in a language like C, as a straight iteration). This means that when there is no sharing of nodes by parents, the algorithm tends to iteratively zip up chains of single-parent links without pushing stack frames.

7 Extended Example

We can see the sequences of steps taken by the algorithm on a complete example in Fig. 4. Part 4(a) shows the initial redex, which is $(\lambda x.(x(\lambda y.x(uy)))(\lambda y.x(uy)))\ t$, where the $(\lambda y.x(uy))$ subterm is shared, and t and u are arbitrary, unspecified subterms with no free occurrences of x or y. To help motivate the point of the algorithm, imagine that the sub-terms t and u are enormous—things we'd like to avoid copying or searching—and that the λx node has other parents besides application 1—so we cannot blindly mutate it at will, without corrupting what the other parents see. (If the λx node *doesn't* have other parents, then the single-reference fast-path described in the previous section applies, and we *are* allowed to mutate the term, for a very fast reduction.)

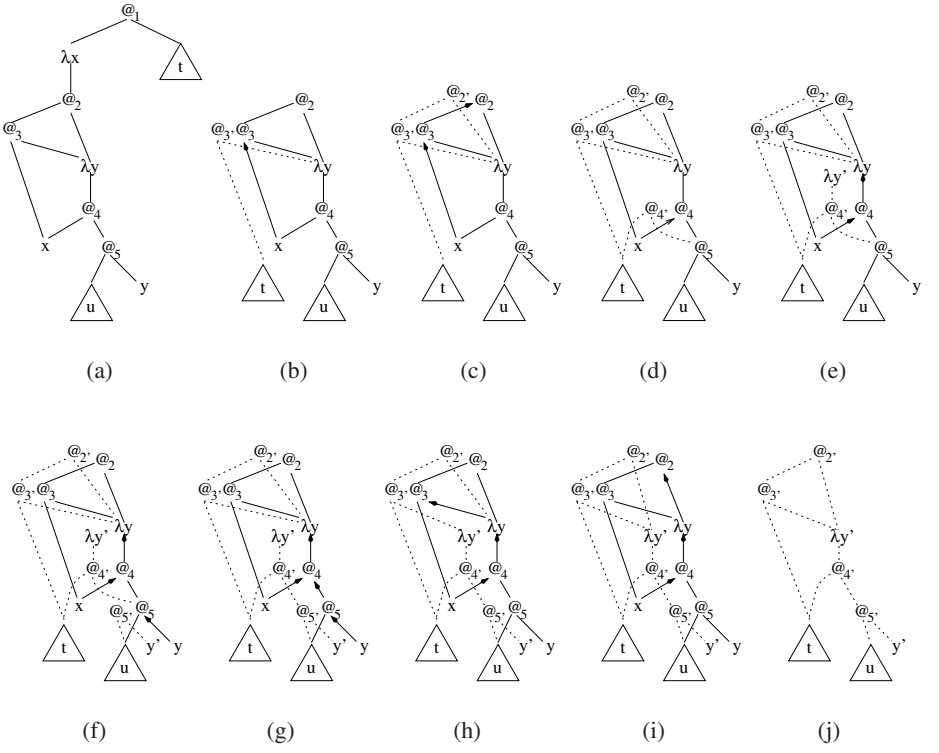


Fig. 4. A trace of a bottom-up reduction of the term $(\lambda x.(x(\lambda y.x(uy)))(\lambda y.x(uy)))t$, where the $(\lambda y.x(uy))$ term is shared, and sub-terms t and u are not specified

In the following subfigure, 4(b), we focus in on the body of the λ-expression being reduced. We iterate over the parents of its variable-reference x, doing an $x \mapsto t$ upcopy; this is the redex-mandated substitution that kicks off the entire reduction. The first parent of x is application 3, which is copied, producing application $3'$, which has function child t instead of the variable reference x, but has the same argument child as the original application 3, namely the λy term. The copy $3'$ is saved away in 3's cache slot, in case we upcopy into 3 from its argument child in the future.

Once we've made a copy of a parent node, we must recursively perform an upcopy for it. That is, we propagate a $3 \mapsto 3'$ upcopy to the parents of application 3. There is only one such parent, application 2. In subfigure 4(c), we see the result of this upcopy: the application $2'$ is created, with function child $3'$ instead of 3; the argument child, λy, is carried over from the original application 2. Again, application $2'$ is saved away in the cache slot of application 2.

Application 2 is the root of the upcopy DAG, so once it has been copied, control returns to application 3 and its $3 \mapsto 3'$ upcopy. Application 3 has only one parent, so it is done. Control returns to x and its $x \mapsto t$ upcopy, which proceeds to propagate upwards to the second parent of x, application 4.

We see the result of copying application 4 in subfigure 4(d). The new node is $4'$, which has function child t where 4 has x; $4'$ shares its argument child, application 5, with application 4. Once again, the copy $4'$ is saved away in the cache slot of application 4.

Having copied application 4, we recursively trigger a $4 \mapsto 4'$ upcopy, which proceeds upwards to the sole parent of application 4. We make a copy of λy, allocating a fresh variable y', with the new body $4'$. This is shown in subfigure 4(e).

Since the new $\lambda y'$ term binds a fresh variable, while processing the λy term we must recursively trigger a $y \mapsto y'$ upcopy, which begins in subfigure 4(f). We iterate through the parents of variable-reference y, of which there is only one: application 5. This is copied, mapping child y to replacement y' and sharing function child u. The result, $5'$, is saved away in the cache slot of application 5.

We then recursively trigger a $5 \mapsto 5'$ upcopy through the parents of application 5; there is only one, application 4. Upon examining this parent (subfigure 4(g)), we discover that 4 already has a copy, $4'$, occupying its cache slot. Rather than create a second, new copy of 4, we simply mutate the existing copy so that its argument child is the new term $5'$. Mutating rather than freshly allocating means the upcopy proceeds no further; responsibility for proceeding upwards from 4 was handled by the thread of computation that first encountered it and created $4'$. So control returns to application 5, which has no more parents, and then to y, who also has no more parents, so control finally returns to the λy term that kicked off the $y \mapsto y'$ copy back in subfigure 4(f).

In subfigure 4(h), the λy term, having produced its copy $\lambda y'$, continues the upcopy by iterating across its parents, recursively doing a $\lambda y \mapsto \lambda y'$ upcopy. The first such parent is application 3, which has already been copied, so it simply mutates its copy to have argument child $\lambda y'$ and returns immediately.

The second parent is application 2, which is handled in exactly the same way in subfigure 4(i). The λy term has no more parents, so it returns control to application 4, who has no more parents, and so returns control to variable reference x. Since x has no more parents, we are done. The answer is application $2'$, which is shown in subfigure 4(j). We can change all references to application 1 in the DAG to point, instead, to application $2'$, and then deallocate 1. Depending on whether or not the children of application 1 have other parents in the DAG, they may also be eligible for deallocation. This is easily performed with a downwards deallocation pass, removing dead nodes from the parent lists of their children, and then recursing if any child thus becomes completely parentless.

8 Experiments

To gain experience with the algorithm, a pair of Georgia Tech undergraduates implemented three β-reduction algorithms: the bottom-up algorithm (BUBS), a reducer based on the suspension λ-calculus (SLC, see Sec. 9.1), and a simple, base-line reducer, based on the simple top-down, blind-search recursive procedure described in Sec. 1. For a toy client application that would generate many requests for reduction, we then built a pair of simple normalisers (one total and one weak-head) on top of the reducers. We did two independent implementations, the first in SML, and a second, in C; the C implementation gave us tighter control over the algorithm and data structures for the purposes of

	CPU time (ms)			# reductions	
	BUBS	SLC	Simple	BUBS	Tree
(fact 2)	0	10	10	123	180
(fact 3)	0	20	20	188	388
(fact 4)	0	40	∞	286	827
(fact 5)	0	160	∞	509	2045
(fact 6)	10	860	∞	1439	7082
(fact 7)	20	5620	∞	7300	36180
(fact 8)	190	48600	∞	52772	245469
nasty-I	30	740	∞	7300	8664
pearl10	0	N/A	N/A	10	N/A
pearl18	0	N/A	N/A	18	N/A
tree10	0	0	0	1023	1023
tree18	740	2530	1980	262143	262143

Fig. 5. Timings for three different implementations of reduction. The system gave us a measurement precision of 10 ms; an entry of 0ms means below the resolution of the timer—*i.e.*, less than 10ms; a measurement of ∞ means the measurement was halted after several cpu-minutes

measurement. The SLC and simple reducers managed storage in the C implementation with the Boehm-Demers-Weiser garbage collector, version 6.2; the BUBS algorithm requires no garbage collector.

Space limitations restrict us to presenting a single set of comparisons from these tests (Fig. 5). The "fact" entries are factorial computations, with Church-numeral encodings. "Nasty-I" is a 20,152-node tree of S and K combinators that reduces to I. A "tree *i*" entry is a full binary tree of applications, *i* deep, with I combinators at the leaves; a "pearl *i*" is this tree collapsed to a DAG—a linear sequence of *i* application nodes with a single I leaf. We compiled the code with gcc 2.95.4 -g -O2 -Wall and performed the test runs on an 800 MHz PIII (256 KB cache), 128 MB RAM, Debian GNU/Linux 3.0 system. These measurements are fairly minimal; we are currently porting Shao's FLINT [14] system to BUBS to get a more realistic test of the algorithm in actual practice.

One of the striking characteristics of the bottom-up algorithm is not only how fast it is, but how well-behaved it seems to be. The other algorithms we've tried have fast cases, but also other cases that cause them to blow up fairly badly. The bottom-up algorithm reliably turns in good numbers. We conjecture this is the benefit of being able to exploit both sharing and non-sharing as they arise in the DAG. If there's sharing, we benefit from re-using work. If there's no sharing, we can exploit the single-parent fast path. These complementary techniques may combine to help protect the algorithm from being susceptible to particular inputs.

9 Related Work

A tremendous amount of prior work has been carried out exploring different ways to implement β-reduction efficiently. In large part, this is due to β-reduction lying at the

heart of the graph-reduction engines that are used to execute lazy functional languages. The text by Peyton Jones *et al.* [13] summarises this whole area very well.

However, the focus of the lazy-language community is on representations tuned for *execution*, and the technology they have developed is cleverly specialised to serve this need. This means, for example, that it's fair game to fix on a particular reduction order. For example, graph reducers that overwrite nodes rely on their normalisation order to keep the necessary indirection nodes from stacking up pathologically. A compiler, in contrast, is a λ-calculus client that makes reductions in a less predictable order, as analyses reveal opportunities for transformation.

Also, an implementation tuned for execution has license to encode terms, or parts of terms, in a form not available for examination, but, rather, purely for execution. This is precisely what the technique of supercombinator compilation does. Our primary interest at the beginning of this whole effort was instead to work in a setting where the term being reduced is always directly available for examination—again, serving the needs of a compiler, which wants to manipulate and examine terms, not execute them.

9.1 Explicit-Substitution Calculi

One approach to constructing efficient λ-term manipulators is to shift to a language that syntactically encodes environments. The "suspension λ-calculus" developed by Nadathur *et al.* [12] is one such example that has been used with success in theorem provers and compilers. However, these implementations are quite complex, inflict de Bruijn encodings on the client, and their "constant-time" reductions simply shift the burden of the reduction to readback time. In the terms we've defined, these technologies use "blind search" to find the variables being substituted. Also, their use of de Bruijn encodings is a barrier to sharing internal structure: de Bruijn-index references are context dependent. *E.g.*, if a term $\lambda x.y$ appears multiple times underneath a λy parent, the index used for the y reference can vary.

One of the major algorithmic payoffs of these representations, lazy reduction, is not so useful for compilers, which typically must examine all the nodes of a term in the course of processing a program. SLC has been successfully employed inside a compiler to represent Shao's FLINT typed intermediate language [14], but the report on this work makes clear the impressive, if not heroic, degree of engineering required to exploit this technology for compiler internals—the path to good performance couples the core SLC representation with hash consing as well as memoisation of term reductions.

The charm of the bottom-up technique presented here is its simplicity. The data structure is essentially just a simple description of the basic syntax as a datatype, with the single addition of child→parent backpointers. It generalises easily to the richer languages used by real compilers and other language-manipulation systems. It's very simple to examine this data structure during processing; very easy to debug the reduction engine itself. In contrast to more sophisticated and complex representations such as SLC, there are really only two important invariants on the structure: (1) all variables are in scope (any path upwards from a variable reference to the root must go through the variable's binding λ-expression), and (2) uplink backpointers mirror downlink references.

9.2 Director Strings

Director strings [7] are a representation driven by the same core issue that motivates our uplink-DAG representation: they provide a way to guide search when performing β-reduction. In the case of director strings, however, one can do the search top-down. Unfortunately, director strings can impose a quadratic space penalty on our trees. Uplinked λ-DAGs are guaranteed to have linear space requirements. Whether or not the space requirements for a director strings representation will blow up in practice depends, of course, on the terms being manipulated. But the attraction of a linear-space representation is knowing that blow-up is completely impossible.

Like the suspension λ-calculus, director strings have the disadvantage of not being a direct representation of the original term; there is some translation involved in converting a λ-calculus term into a director strings.

Director strings can be an excellent representation choice for graph-reducing normalising engines. Again, we are instead primarily focussed on applications that require fine-grained inter-reduction access to the term structure, such as compilers.

9.3 Optimal λ Reduction

The theory of "optimal λ reduction" [10, 9, 6] (or, OLR), originated by Lévy and Lamping, and developed by Abadi, Asperti, Gonthier, Guerrini, Lawall, Mairson *et al.*, is a body of work that shares much with bottom-up β-reduction. Both represent λ-terms using graph structure, and the key idea of connecting variable-binders directly to value-consumers of the bound variable is present in both frameworks—and for the same reason, namely, from a desire that substitution should be proportional to the number of references to the bound variable, removing the need to blindly search a term looking for these references.

However, the two systems are quite different in their details, in fairly deep ways. Lamping graphs allow *incremental reduction* by means of adding extra "croissant," "bracket" and "fan" nodes to the graph. This exciting alternative model of computation, however, comes with a cost: the greatly increased complexity of the graph structure and its associated operations. In particular, in actual use, the croissant and bracket marks can frequently pile up uselessly along an edge, tying up storage and processing steps. It also makes it difficult to "read" information from the graph structure. As Gonthier, Abadi and Lévy state [6], "it seems fair to say that Lamping's algorithm is rather complicated and obscure." The details of this complexity have prevented OLR-based systems from widespread adoption.

9.4 Two Key Issues: Persistence and Readback

Our comparisons with other techniques have repeatedly invoked the key issues of persistence and readback. Our data structure is not a "persistent" one—performing a reduction inside a term changes the term. If an application needs to keep the old term around, then our algorithm is not a candidate (or, at least, not without some serious surgery). So perhaps it is unfair to compare our algorithm's run times to those of persistent algorithms, such as SLC or director strings.

However, we can turn this around, and claim that the interesting feature of our algorithm is that it *exploits* lack of persistence. If an application doesn't need persistence, it shouldn't have to pay for it. The standard set of choices are invariably persistent; our algorithm provides an alternative design point. (Note that reduction on Lamping graphs is also not persistent, which is, again, either a limitation or a source of efficiency, depending on your point of view.)

The other key, cross-cutting issue is readback. An application that doesn't need to examine term structure in-between reductions has greater flexibility in its requirements. If readback is a requirement, however, then Lamping graphs and the SLC are much less attractive. Readback with our representation is free: one of the pleasant properties of a DAG is that it can be viewed just as easily as a tree; there is no need to convert it.

Thus, bottom-up β-reduction is a technology which is well suited to applications which (1) don't need persistence, but (2) do need fine-grained readback.

10 Conclusion

We certainly are not the first to consider using graph structure to represent terms of the λ-calculus; the ideas go back at least to 1954 [4, 15]. The key point we are making is that two of these ideas work together:

- representing λ-terms as DAGS to allow sharing induced by β-reduction, and
- introducing child$\rightarrow$parent backpointers and $\lambda\rightarrow$variable links to efficiently direct search and construction.

The first idea allows sharing *within* a term, while the second allows sharing *across* a reduction, but they are, in fact, mutually enabling: in order to exploit the backpointers, we need the DAG representation to allow us to build terms without having to replicate the subterm being substituted for the variable. This is the source of speed and space efficiency.

The algorithm is simple and directly represents the term without any obscuring transform, such as combinators, de Bruijn indices or suspensions, a pleasant feature for λ-calculus clients who need to examine the terms. It is also, in the parlance of the graph-reduction community, fully lazy.

Acknowledgements

Bryan Kennedy and Stephen Strickland, undergraduates at Georgia Tech, did the entire implementation and evaluation reported in Sec. 8. We thank, of course, Olivier Danvy. Zhong Shao provided helpful discussions on the suspension λ-calculus. Chris Okasaki and Simon Peyton Jones tutored us on director strings. Harry Mairson and Alan Bawden provided lengthy and patient instruction on the subtleties of optimal lambda reduction and Lamping graphs.

References

1. Andrea Asperti and Stefano Guerrini. *The Optimal Implementation of Functional Programming Languages.* Cambridge University Press, 1999.
2. Henk Barendregt. *The Lambda Calculus.* North Holland, revised edition, 1984.
3. Alan Bawden. Personal communication, November 2002. Alan wrote the compiler, a toy exercise for Scheme, sometime in the late 1980's.
4. N. Bourbaki. *Théorie des ensembles.* Hermann & C. Editeurs, 1954.
5. Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.
6. Georges Gonthier, Martín Abadi and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–26, January 1992.
7. J. R. Kennaway and M. R. Sleep. Director strings as combinators. ACM Transactions on Programming Languages and Systems, 10, pages 602–626, (October 1988).
8. Richard A. Kelsey. A correspondence between continuation-passing style and static single assignment form. In *ACM SIGPLAN Workshop on Intermediate Representations*, *SIGPLAN Notices*, vol. 30, no. 3, pages 13–22, January 1995.
9. John Lamping. An algorithm for optimal lambda-calculus reduction. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 16–30, January 1990.
10. Jean-Jacques Lévy. *Réductions Correctes et Optimales dans le Lambda-calcul.* Ph.D. thesis, Université Paris VII, 1978.
11. R. Milner, M. Tofte, R. Harper, D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.
12. Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science* 198(1–2):49–98, May 1998.
13. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.
14. Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming Languages*, September 1998.
15. C. P. Wadsworth. *Semantics and pragmatics of the lambda-calculus.* PhD dissertation, Oxford University, 1971.

BI Hyperdoctrines and Higher-Order Separation Logic

Bodil Biering, Lars Birkedal*, and Noah Torp-Smith*

Department of Theoretical Computer Science,
IT University of Copenhagen, Denmark
{biering, birkedal, noah}@itu.dk

Abstract. We present a precise correspondence between separation logic and a new simple notion of *predicate* BI, extending the earlier correspondence given between part of separation logic and *propositional* BI [14]. Moreover, we introduce the notion of a BI hyperdoctrine and show that it soundly models classical and intuitionistic first- and higher-order predicate BI, and use it to show that we may easily extend separation logic to *higher-order*. We argue that the given correspondence may be of import for formalizations of separation logic.

1 Introduction

Separation logic [20, 19, 5, 6, 22, 9, 2] is a Hoare-style program logic, and variants of it have been applied to prove correct interesting pointer algorithms such as copying a dag, disposing a graph, the Schorr-Waite graph algorithm, and Cheney's copying garbage collector. Different extensions of core separation logic were employed to conduct these proofs. For example, Yang [21] extended the core logic with lists and trees, and in [2] the logic included finite sets and relations. Thus it is natural to ask whether one has to make a new extension of separation logic for every proof one wants to make [17]. This would be unfortunate for formal verification of proofs in separation logic since it would make the enterprise of formal verification burdensome and dubious. We argue that there is a natural single underlying logic in which it is possible to *define* the various extensions and prove the expected properties thereof; this is then the single logic that should be employed for formal verification.

Part of the pointer model of separation logic, namely that given by heaps (but not stacks), has been related to *propositional* BI, the logic of bunched implications introduced by O'Hearn and Pym [12]. In this paper we show how the correspondence may be extended to a precise correspondence between all of the pointer model (including stacks) and a simple notion of *predicate* BI. We introduce the notion of a *BI hyperdoctrine*, a simple extension of Lawvere's notion of hyperdoctrine [8], and show that it soundly models predicate BI. We consider a different notion of predicate BI than that of [15, 16], which has a BI structure on contexts. However, we believe that our notion of predicate BI with its class of BI hyperdoctrine models is the right one for separation logic (Pym aimed to

* Partially supported by Danish Technical Research Council Grant 56-00-0309.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 233–247, 2005.

model mulitiplicative quantifiers; separation logic only uses additive quantifiers). To make this point, we show that the pointer model of separation logic exactly corresponds to the interpretation of predicate BI in a simple BI hyperdoctrine. This correspondence also allows us to see that it is simple to extend separation logic to *higher-order* separation logic. We explain this extension and suggest that it may be useful for program proving.

Before proceeding with the technical development we give an intuitive justification of the use of BI hyperdoctrines to model higher-order predicate BI. A powerful way of obtaining models of BI is by means of functor categories (presheaves), using Day's construction to obtain a doubly-closed structure on the functor category [14]. Such functor categories can be used to model *propositional* BI in two different senses: In the first sense, one models *provability*, entailment between propositions, and it works because the lattice of subobjects of the terminal object in such functor categories form a BI algebra (a doubly cartesian closed preorder). In the second sense, one models *proofs*, and it works because the whole functor category is doubly cartesian closed. Here we seek models of provability of *predicate* BI. Since the considered functor categories are toposes and hence model higher-order predicate logic, one might think that a straightforward extension is possible. But, alas, it is not the case. In general, for this to work, *every* lattice of subobjects (for any object, not only for the terminal object) should be a BI algebra and, moreover, to model substitution correctly the BI algebra structure should be preserved by pulling back along any morphism. We show that this can only be the case if the BI algebra structure is trivial, that is, coincides with the cartesian structure (see Theorem 7). Our theorem holds for any topos, not just for the functor categories considered earlier. Hence we need to consider a wider class of models for predicate BI than just toposes and this justifies the notion of a BI hyperdoctrine. The intuitive reason that BI hyperdoctrines work, is that predicates are not required to be modeled by subobjects, they can be something more general. Another important point of BI hyperdoctrines is that they are easy to come by: given any complete BI algebra B, we can define a canonical BI hyperdoctrine in which predicates are modeled as B-valued functions; we explain this in detail in Example 6.

The remainder of this paper is organized as follows. In Section 2 we recall the notion of a (first-order) hyperdoctrine and explain how it soundly models predicate logic. We then define the concept of a (first-order) BI hyperdoctrine and explain how it soundly models predicate BI. In Section 3 we briefly recall the standard pointer model of separation logic and show how it can be construed as a first-order BI hyperdoctrine. In Section 4 we discuss some consequences for separation logic, and in particular, we use the higher-order logic to give logical characterizations of interesting classes of predicates. Finally, we conclude in Section 5.

2 BI Hyperdoctrines

In this section we introduce Lawvere's notion of a *hyperdoctrine* [8] and briefly recall how it can be used to model intuitionistic and classical first- and higher-

order predicate logic (see, for example, [13] and [7] for more explanations than can be included here). We then define the notion of a BI hyperdoctrine, which is a straightforward extension of the standard notion of hyperdoctrine, and explain how it can be used to model predicate BI logic.

Hyperdoctrines. A first-order hyperdoctrine is a categorical structure tailored to model first-order predicate logic with equality. The structure has a base category $\mathcal{C}$ for modeling the types and terms, and a $\mathcal{C}$-indexed category $\mathcal{P}$ for modeling formulas.

Definition 1 (First-order hyperdoctrines). *Let $\mathcal{C}$ be a category with finite products. A* first-order hyperdoctrine *$\mathcal{P}$ over $\mathcal{C}$ is a contravariant functor $\mathcal{P} : \mathcal{C}^{op} \to$ Poset from $\mathcal{C}$ into the category of partially ordered sets and monotone functions, with the following properties.*

1. *For each object X, the partially ordered set $\mathcal{P}(X)$ is a Heyting algebra.*
2. *For each morphism $f : X \to Y$ in $\mathcal{C}$, the monotone function $\mathcal{P}(f) : \mathcal{P}(Y) \to \mathcal{P}(X)$ is a Heyting algebra homomorphism.*
3. *For each diagonal morphism $\Delta_X : X \to X \times X$ in $\mathcal{C}$, the left adjoint to $\mathcal{P}(\Delta_X)$ at the top element $\top \in \mathcal{P}(X)$ exists. In other words, there is an element $=_X$ of $\mathcal{P}(X \times X)$ satisfying that for all $A \in \mathcal{P}(X \times X)$,*

$$\top \leq \mathcal{P}(\Delta_X)(A) \quad \textit{iff} \quad =_X \leq A.$$

4. *For each product projection $\pi : \Gamma \times X \to \Gamma$ in $\mathcal{C}$, the monotone function $\mathcal{P}(\pi) : \mathcal{P}(\Gamma) \to \mathcal{P}(\Gamma \times X)$ has both a left adjoint $(\exists X)_\Gamma$ and a right adjoint $(\forall X)_\Gamma$:*

$$A \leq \mathcal{P}(\pi)(A') \quad \textit{if and only if} \quad (\exists X)_\Gamma(A) \leq A'$$

$$\mathcal{P}(\pi)(A') \leq A \quad \textit{if and only if} \quad A' \leq (\forall X)_\Gamma(A).$$

Moreover, these adjoints are natural in Γ, i.e., given $s : \Gamma \to \Gamma'$ in $\mathcal{C}$, we have

$$\begin{array}{ccc} \mathcal{P}(\Gamma' \times X) & \xrightarrow{\mathcal{P}(s \times \mathrm{id}_X)} & \mathcal{P}(\Gamma \times X) \\ {\scriptstyle (\exists X)_{\Gamma'}}\downarrow & & \downarrow{\scriptstyle (\exists X)_\Gamma} \\ \mathcal{P}(\Gamma') & \xrightarrow[\mathcal{P}(s)]{} & \mathcal{P}(\Gamma) \end{array} \qquad \begin{array}{ccc} \mathcal{P}(\Gamma' \times X) & \xrightarrow{\mathcal{P}(s \times \mathrm{id}_X)} & \mathcal{P}(\Gamma \times X) \\ {\scriptstyle (\forall X)_{\Gamma'}}\downarrow & & \downarrow{\scriptstyle (\forall X)_\Gamma} \\ \mathcal{P}(\Gamma') & \xrightarrow[\mathcal{P}(s)]{} & \mathcal{P}(\Gamma). \end{array}$$

The elements of $\mathcal{P}(X)$, where X ranges over objects of $\mathcal{C}$, will be referred to as $\mathcal{P}$-predicates.

Interpretation of First-Order Logic in a First-Order Hyperdoctrine. Given a (first order) signature with types X, function symbols $f : X_1, \ldots, X_n \to X$, and relation symbols $R \subset X_1, \ldots, X_n$, a *structure* for the signature in a first-order hyperdoctrine $\mathcal{P}$ over $\mathcal{C}$ assigns an object $[\![X]\!]$ in $\mathcal{C}$ to each type, a morphism $[\![f]\!] : [\![X_1]\!] \times \cdots \times [\![X_n]\!] \to [\![X]\!]$ to each function symbol, and a $\mathcal{P}$-predicate

$[\![R]\!] \in \mathcal{P}([\![X_1]\!] \times \cdots \times [\![X_n]\!])$ to each relation symbol. Any term t over the signature, with free variables in $\Gamma = \{x_1 : X_1, \ldots, x_n : X_n\}$ and of type X say, is interpreted as a morphism $[\![t]\!] : [\![\Gamma]\!] \to [\![X]\!]$, where $[\![\Gamma]\!] = [\![X_1]\!] \times \cdots \times [\![X_n]\!]$, by induction on the structure of t (in the standard manner in which terms are interpreted in categories).

Each formula ϕ with free variables in Γ is interpreted as a $\mathcal{P}$-predicate $[\![\phi]\!] \in \mathcal{P}([\![\Gamma]\!])$ by induction on the structure of ϕ using the properties given in Definition 1. For atomic formulas $R(t_1, \ldots, t_n)$, the interpretation is given by $\mathcal{P}(\langle [\![t_1]\!], \ldots, [\![t_n]\!] \rangle)([\![R]\!])$. In particular, the atomic formula $t =_X t'$ is interpreted by the $\mathcal{P}$-predicate $\mathcal{P}(\langle [\![t]\!], [\![t']\!] \rangle)(=_{[\![X]\!]})$. The interpretation of other formulas is given by structural induction. Assume ϕ, ϕ' are formulas with free variables in Γ and that ψ is a formula with free variables in $\Gamma \cup \{x : X\}$. Then,

$$
\begin{array}{ll}
[\![\top]\!] = \top_H & [\![\phi \wedge \phi']\!] = [\![\phi]\!] \wedge_H [\![\phi']\!] \\
[\![\bot]\!] = \bot_H & [\![\phi \vee \phi']\!] = [\![\phi]\!] \vee_H [\![\phi']\!] \\
 & [\![\phi \to \phi']\!] = [\![\phi]\!] \to_H [\![\phi']\!]
\end{array}
$$

$$
\begin{array}{l}
[\![\forall x : X.\psi]\!] = (\forall [\![X]\!])_{[\![\Gamma]\!]}([\![\psi]\!]) \in \mathcal{P}([\![\Gamma]\!]) \\
[\![\exists x : X.\psi]\!] = (\exists [\![X]\!])_{[\![\Gamma]\!]}([\![\psi]\!]) \in \mathcal{P}([\![\Gamma]\!]),
\end{array}
$$

where $\wedge_H, \vee_H$, etc., is the Heyting algebra structure on $\mathcal{P}([\![\Gamma]\!])$.

We say that a formula ϕ with free variables in Γ is *satisfied* if $[\![\phi]\!]$ is the top element of $\mathcal{P}([\![\Gamma]\!])$. This notion of satisfaction is *sound* for intuitionistic predicate logic, in the sense that all provable formulas are satisfied. Moreover, it is also *complete* in the sense that a formula is provable if it is satisfied in all structures in first-order hyperdoctrines. A first-order hyperdoctrine $\mathcal{P}$ is sound for *classical* predicate logic in case all the fibres $\mathcal{P}(X)$ are Boolean algebras and all the reindexing functions $\mathcal{P}(f)$ are Boolean algebra homomorphisms.

Definition 2 (Hyperdoctrines). *A (general)* hyperdoctrine *is a first-order hyperdoctrine with the following additional properties: $\mathcal{C}$ is cartesian closed, and there is a Heyting algebra H and a natural bijection $\Theta_X : Obj(\mathcal{P}(X)) \simeq \mathcal{C}(X, H)$.*

A hyperdoctrine is sound for higher-order intuitionistic predicate logic: the Heyting algebra H is used to interpret the type, call it prop, of propositions and higher types (e.g., prop^X, the type for predicates over X), are interpreted by exponentials in $\mathcal{C}$. The natural bijection Θ_X is used to interpret substitution of formulas in formulas: Suppose ϕ is a formula with a free variable q of type prop and with remaining free variables in Γ, and that ψ is a formula with free variables in Γ. Then $[\![\psi]\!] \in \mathcal{P}([\![\Gamma]\!])$, $[\![\phi]\!] \in \mathcal{P}([\![\Gamma]\!] \times H)$, and $\phi[\psi/q]$ (ϕ with ψ substituted in for q) is interpreted by $\mathcal{P}(\langle \mathrm{id}, \Theta([\![\psi]\!]) \rangle)([\![\phi]\!])$. For more details see, e.g., [13].

Again it is the case that a hyperdoctrine $\mathcal{P}$ is sound for *classical* higher-order predicate logic in case all the fibres $\mathcal{P}(X)$ are Boolean algebras and all the reindexing functions $\mathcal{P}(f)$ are Boolean algebra homomorphisms.

Example 3 (Canonical hyperdoctrine over a topos). *Let $\mathcal{E}$ be a topos. It is well-known that $\mathcal{E}$ models higher-order predicate logic, by interpreting types as*

objects in $\mathcal{E}$, terms as morphisms in $\mathcal{E}$ and predicates as subobjects in $\mathcal{E}$. The topos $\mathcal{E}$ induces a canonical $\mathcal{E}$-indexed hyperdoctrine $\mathrm{Sub}_{\mathcal{E}} : \mathcal{E}^{op} \to \mathrm{Poset}$*, which maps an object X in $\mathcal{E}$ to the poset of subobjects of X in $\mathcal{E}$ and a morphisms $f : X \to Y$ to the pullback functor $f^* : \mathrm{Sub}(Y) \to \mathrm{Sub}(X)$. Then the standard interpretation of predicate logic in $\mathcal{E}$ coincides with the interpretation of predicate logic in the hyperdoctrine* $\mathrm{Sub}_{\mathcal{E}}$*. Compared to the standard interpretation in toposes, however, hyperdoctrines allow that predicates are not always modeled by subobjects but can come from some other universe. Thus hyperdoctrines describe a wider class of models than toposes do.*

BI Hyperdoctrines. Recall that a Heyting algebra is a bi-cartesian closed partial order, i.e., a partial order, which, when considered as a category, is cartesian closed ($\top$, $\wedge$, $\to$) and has finite coproducts ($\bot$, $\vee$). Further recall that a *BI algebra* is a Heyting algebra, which has an additional symmetric monoidal closed structure (I, $*$, $-\!\!*$) [15].

We now present a straightforward extension of (first-order) hyperdoctrines, which models first and higher-order predicate BI.

Definition 4 (BI Hyperdoctrines).

- *A first-order hyperdoctrine $\mathcal{P}$ over $\mathcal{C}$ is a* first-order BI hyperdoctrine *in case all the fibres $\mathcal{P}(X)$ are BI algebras and all the reindexing functions $\mathcal{P}(f)$ are BI algebra homomorphisms.*
- *A* BI hyperdoctrine *is a first-order BI hyperdoctrine with the additional properties that $\mathcal{C}$ is cartesian closed, and there is a BI algebra B and a natural bijection $\Theta_X : Obj(\mathcal{P}(X)) \simeq \mathcal{C}(X, B)$.*

First-order predicate BI is first-order predicate logic with equality, extended with formulas I, $\phi * \psi$, $\phi -\!\!* \psi$ satisfying the following rules (in any context Γ including the free variables of the formulas):

$$(\phi * \psi) * \theta \vdash_\Gamma \phi * (\psi * \theta) \qquad \phi * (\psi * \theta) \vdash_\Gamma (\phi * \psi) * \theta \qquad \vdash_\Gamma \phi \leftrightarrow \phi * \mathrm{I}$$

$$\phi * \psi \vdash_\Gamma \psi * \phi \qquad \frac{\phi \vdash_\Gamma \psi \quad \theta \vdash_\Gamma \omega}{\phi * \theta \vdash_\Gamma \psi * \omega} \qquad \frac{\phi * \psi \vdash_\Gamma \theta}{\phi \vdash_\Gamma \psi -\!\!* \theta}$$

Our notion of predicate BI should not be confused with the one presented in [15]; the latter seeks to also include a BI structure on contexts but we do not attempt to do that here, since that is not what is used in separation logic. In particular, weakening at the level of variables is always allowed:

$$\frac{\phi \vdash_\Gamma \psi}{\phi \vdash_{\Gamma \cup \{x:X\}} \psi}$$

We can interpret first-order predicate BI in a first-order BI hyperdoctrine simply by extending the interpretation of first-order logic in first-order hyperdoctrine given above by:

$$\begin{aligned} [\![\mathrm{I}]\!] &= \mathrm{I}_B \\ [\![\phi * \psi]\!] &= [\![\phi]\!] *_B [\![\psi]\!] \\ [\![\phi -\!\!* \psi]\!] &= [\![\phi]\!] -\!\!*_B [\![\psi]\!], \end{aligned}$$

where I_B, $*_B$ and $-\!\!*_B$ is the monoidal closed structure in the BI algebra $\mathcal{P}(\llbracket \Gamma \rrbracket)$. We then have:

Theorem 5. *The interpretation of first-order predicate BI given above is sound and complete.*

Likewise, BI hyperdoctrines form sound and complete models for higher-order predicate BI. Of course, a (first-order) BI hyperdoctrine is sound for classical BI in case all the fibres $\mathcal{P}(X)$ are Boolean BI algebras and all the reindexing functions $\mathcal{P}(f)$ are Boolean BI algebra homomorphisms. Here is a canonical example of a BI hyperdoctrine.

Example 6 (BI hyperdoctrine over a complete BI algebra). *Let B be a complete BI algebra, i.e., it has all joins and meets. It determines a BI hyperdoctrine over the category* **Set** *as follows. For each set X, let $\mathcal{P}(X) = B^X$, the set of functions from X to B, ordered pointwise. Given $f : X \to Y$, $\mathcal{P}(f) : B^Y \to B^X$ is the BI algebra homomorphism given by composition with f. For example if $s, t \in \mathcal{P}(Y)$, i.e., $s, t : Y \to B$, then $\mathcal{P}(f)(s) = s \circ f : X \to B$ and $s * t$ is defined pointwise as $(s * t)(y) = s(y) * t(y)$. Equality predicates $=_X$ in $B^{X \times X}$ are given by*

$$=_X (x, x') \stackrel{def}{=} \begin{cases} \top \text{ if } x = x' \\ \bot \text{ if } x \neq x' \end{cases}$$

where $\top$ and $\bot$ are the greatest and least elements of B, respectively. The quantifiers use set-indexed joins ($\bigvee$) and meets ($\bigwedge$). Specifically, given $A \in B^{\Gamma \times X}$ one has

$$(\exists X)_\Gamma(A) \stackrel{def}{=} \lambda i \in \Gamma. \bigvee_{x \in X} A(i, x) \qquad (\forall X)_\Gamma(A) \stackrel{def}{=} \lambda i \in \Gamma. \bigwedge_{x \in X} A(i, x)$$

in B^Γ. The conditions in Definition 2 are trivially satisfied (Θ is the identity).

There are plenty of examples of complete BI algebras: for any Grothendieck topos $\mathcal{E}$ with an additional symmetric monoidal closed structure, $\mathrm{Sub}_\mathcal{E}(1)$ is a complete BI algebra, and for any monoidal category $\mathcal{C}$ such that the monoid is cover preserving w.r.t. the Grothendieck topology J, $\mathrm{Sub}_{\mathrm{Sh}(\mathcal{C},J)}(1)$ is a complete BI algebra [1, 14].

The following theorem shows that to get interesting models of higher-order predicate BI, it does not suffice to consider BI hyperdoctrines arising as the canoncial hyperdoctrine over a topos (as in Example 3). Indeed this is the reason for introducing the more general BI hyperdoctrines. For reasons of space, we omit the proof in this exposition.

Theorem 7. *Let $\mathcal{E}$ be a topos and suppose* $\mathrm{Sub}_\mathcal{E} : \mathcal{E}^{op} \to$ Poset *is a BI hyperdoctrine. Then the BI structure on each lattice $\mathrm{Sub}_\mathcal{E}(X)$ is trivial, i.e., for all $\varphi, \psi \in \mathrm{Sub}_\mathcal{E}(X)$, $\varphi * \psi \leftrightarrow \varphi \wedge \psi$.*

3 Separation Logic Modeled by BI-Hyperdoctrines

We briefly recall the standard pointer model of separation logic (for a more thorough presentation see, for instance, [20]) and then show how it can be construed as a BI hyperdoctrine over **Set**.

The core assertion language of separation logic (which we will henceforth also call separation logic) is often defined as follows. There is a single type Val of values. Terms t are defined by a grammar

$$t ::= x \mid n \mid t + t \mid t - t \mid \cdots,$$

where $n : \mathrm{Val}$ are constants for all integers n. Formulas, also called assertions, are defined by

$$\phi ::= \top \mid \bot \mid t = t \mid t \mapsto t \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \to \phi \mid \phi * \phi \mid \phi \mathrel{-\!\!*} \phi \mid \mathbf{emp} \mid \forall x.\phi \mid \exists x.\phi$$

The symbol **emp** is used in separation logic for the unit of BI.

Note that the above is just another way of defining a signature (specification of types, function symbols and predicate symbols) for first-order predicate BI with a single type Val, function symbols $+, -, \ldots : \mathrm{Val}, \mathrm{Val} \to \mathrm{Val}$, constants $n : \mathrm{Val}$, and relation symbol $\mapsto \subseteq \mathrm{Val}, \mathrm{Val}$.

The Pointer Model. The standard pointer model of separation logic is usually presented as follows. It consists of a set $[\![\mathrm{Val}]\!]$ interpreting the type Val and a set $[\![\mathrm{Loc}]\!]$ of locations such that $[\![\mathrm{Loc}]\!] \subseteq [\![\mathrm{Val}]\!]$ and binary functions on $[\![\mathrm{Val}]\!]$ interpreting the function symbols $+, -$. The set $H = [\![\mathrm{Loc}]\!] \rightharpoonup_{fin} [\![\mathrm{Val}]\!]$ of finite partial functions from $[\![\mathrm{Loc}]\!]$ to $[\![\mathrm{Val}]\!]$, ordered discretely, is referred to as the set of *heaps*. The set of heaps has a partial binary operation $*$ defined by

$$h_1 * h_2 = \begin{cases} h_1 \cup h_2 & \text{if } h_1 \# h_2 \\ \text{undefined} & \text{otherwise,} \end{cases}$$

where $\#$ is the binary relation on heaps defined by $h_1 \# h_2$ iff $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$. The interpretation of the relation $\mapsto \subseteq [\![\mathrm{Val}]\!] \times [\![\mathrm{Val}]\!]$ is the subset of singleton heaps, that is, for $h \in H$, $h \in \mapsto$ iff $h = \{(v_1, v_2)\}$ for some values v_1, v_2. To define the standard interpretation of terms and formulas, one assumes a partial function $s : \mathrm{Var} \rightharpoonup_{fin} [\![\mathrm{Val}]\!]$, called a stack (also called a store in the literature). The interpretation of terms depends on the stack and is defined by

$$\begin{aligned} [\![x]\!]s &= s(x) \\ [\![n]\!]s &= [\![n]\!] \\ [\![t_1 \pm t_2]\!]s &= [\![t_1]\!]s \pm [\![t_2]\!]s \end{aligned}$$

The interpretation of formulas is standardly given by a forcing relation $s, h \Vdash \phi$, where $\mathrm{FV}(\phi) \subseteq \mathrm{dom}(s)$, as follows

$$
\begin{array}{lll}
s,h \Vdash t_1 = t_2 & \text{iff} & [\![t_1]\!]s = [\![t_2]\!]s \\
s,h \Vdash t_1 \mapsto t_2 & \text{iff} & \mathrm{dom}(h) = \{[\![t_1]\!]s\} \text{ and } h([\![t_1]\!]s) = [\![t_2]\!]s \\
s,h \Vdash \mathbf{emp} & \text{iff} & h = \emptyset \\
s,h \Vdash \top & & \text{always} \\
s,h \Vdash \bot & & \text{never} \\
s,h \Vdash \phi * \psi & \text{iff} & \text{there exists } h_1, h_2 \in H.\, h_1 * h_2 = h \text{ and} \\
 & & s,h_1 \Vdash \phi \text{ and } s,h_2 \Vdash \psi \\
s,h \Vdash \phi \mathrel{-\!\!*} \psi & \text{iff} & \text{for all } h', h' \# h \text{ and } s,h' \Vdash \phi \text{ implies } s, h * h' \Vdash \psi \\
s,h \Vdash \phi \vee \psi & \text{iff} & s,h \Vdash \phi \text{ or } s,h \Vdash \psi \\
s,h \Vdash \phi \wedge \psi & \text{iff} & s,h \Vdash \phi \text{ and } s,h \Vdash \psi \\
s,h \Vdash \phi \to \psi & \text{iff} & s,h \Vdash \phi \text{ implies } s,h \Vdash \psi \\
s,h \Vdash \forall x.\phi & \text{iff} & \text{for all } v \in [\![\mathrm{Val}]\!].s[x \mapsto v], h \Vdash \phi \\
s,h \Vdash \exists x.\phi & \text{iff} & \text{there exists } v \in [\![\mathrm{Val}]\!].s[x \mapsto v], h \Vdash \phi
\end{array}
$$

We now show how this pointer model is an instance of a BI-hyperdoctrine of a complete Boolean BI algebra (cf. Example 6).

The Pointer Model as a BI Hyperdoctrine. Let $(H_\bot, *)$ be the discretely ordered set of heaps with a bottom element added to represent undefined, and let $* : H_\bot \times H_\bot \to H_\bot$ be the total extension of $* : H \times H \rightharpoonup H$ satisfying $\bot * h = h * \bot = \bot$, for all $h \in H_\bot$. This defines a partially ordered commutative monoid with the empty heap $\{\}$ as the unit for $*$. The powerset of H, $\mathcal{P}(H)$ (without $\bot$) is a complete Boolean BI algebra, ordered by inclusion and with monoidal closed structure given by (for $U, V \in \mathcal{P}(H)$):

- I is $\{\emptyset\}$
- $U * V := \{h * h' \mid h \in U \wedge h' \in V\} \setminus \{\bot\}$
- $U \mathrel{-\!\!*} V := \bigcup\{W \subseteq H \mid (W * U) \subseteq V\}$.

It can easily be verified directly that this defines a complete Boolean BI algebra; it also follows from more abstract arguments in [14, 1].

Let S be the BI hyperdoctrine induced by the complete Boolean BI algebra $\mathcal{P}(H)$ as in Example 6. To show that the interpretation of separation logic in this BI hyperdoctrine exactly corresponds to the standard pointer model presented above we spell out the interpretation of separation logic in S.

A term t in a context $\Gamma = \{x_1 : \mathrm{Val}, \ldots, x_n : \mathrm{Val}\}$ is interpreted as a morphism between sets:

- $[\![x_i : \mathrm{Val}]\!] = \pi_i$, where $\pi_i : \mathrm{Val}^n \to \mathrm{Val}$ is the i'th projection,
- $[\![n]\!]$ is the map $[\![n]\!] : [\![\Gamma]\!] \to 1 \to [\![\mathrm{Val}]\!]$ which sends the unique element of the one-point set 1 to $[\![n]\!]$,
- $[\![t_1 \pm t_2]\!] = [\![t_1]\!] \pm [\![t_2]\!] : [\![\Gamma]\!] \to [\![\mathrm{Val}]\!] \times [\![\mathrm{Val}]\!] \to [\![\mathrm{Val}]\!]$, where $[\![t_i]\!] : [\![\Gamma]\!] \to [\![\mathrm{Val}]\!]$, for $i = 1, 2$.

The interpretation of a formula ϕ in a context $\Gamma = \{x_1 : \mathrm{Val}, \ldots, x_n : \mathrm{Val}\}$ is given inductively as follows. Let $I = [\![\mathrm{Val}]\!] \times \cdots \times [\![\mathrm{Val}]\!] = [\![\mathrm{Val}]\!]^n$ and write $\overline{v}$ for elements of I. Then ϕ is interpreted as an element of $\mathcal{P}(I)$ as follows:

$$
\begin{array}{ll}
[\![t_1 \mapsto t_2]\!](\overline{\mathrm{v}}) & = \{h \mid \mathrm{dom}(h) = \{[\![t_1]\!](\overline{\mathrm{v}})\} \text{ and } h([\![t_1]\!](\overline{\mathrm{v}})) = [\![t_2]\!](\overline{\mathrm{v}})\} \\
[\![t_1 = t_2]\!](\overline{\mathrm{v}}) & = H \text{ if } [\![t_1]\!](\overline{\mathrm{v}}) = [\![t_2]\!](\overline{\mathrm{v}}),\ \emptyset \text{ otherwise} \\
[\![\top]\!](*) & = H \\
[\![\bot]\!](*) & = \emptyset \\
[\![\mathbf{emp}]\!](*) & = \{h \mid \mathrm{dom}(h) = \emptyset\} \\
[\![\phi \wedge \psi]\!](\overline{\mathrm{v}}) & = [\![\phi]\!](\overline{\mathrm{v}}) \cap [\![\psi]\!](\overline{\mathrm{v}}) \\
[\![\phi \vee \psi]\!](\overline{\mathrm{v}}) & = [\![\phi]\!](\overline{\mathrm{v}}) \cup [\![\psi]\!](\overline{\mathrm{v}}) \\
[\![\phi \to \psi]\!](\overline{\mathrm{v}}) & = \{h \mid h \in [\![\phi]\!](\overline{\mathrm{v}}) \text{ implies } h \in [\![\psi]\!](\overline{\mathrm{v}})\} \\
[\![\phi * \psi]\!](\overline{\mathrm{v}}) & = [\![\phi]\!](\overline{\mathrm{v}}) * [\![\psi]\!](\overline{\mathrm{v}}) \\
 & = \{h_1 * h_2 \mid h_1 \in [\![\phi]\!](\overline{\mathrm{v}}) \text{ and } h_2 \in [\![\psi]\!](\overline{\mathrm{v}})\} \setminus \{\bot\} \\
[\![\phi \mathrel{-\!\!*} \psi]\!](\overline{\mathrm{v}}) & = [\![\phi]\!](\overline{\mathrm{v}}) \mathrel{-\!\!*} [\![\psi]\!](\overline{\mathrm{v}}) \\
 & = \{h \mid [\![\phi]\!](\overline{\mathrm{v}}) * \{h\} \subseteq [\![\psi]\!](\overline{\mathrm{v}})\} \\
[\![\forall x : \mathrm{Val}\,.\phi]\!](\overline{\mathrm{v}}) & = \bigcap_{v_x \in [\![\mathrm{Val}]\!]} ([\![\phi]\!](v_x, \overline{\mathrm{v}})) \\
[\![\exists x : \mathrm{Val}\,.\phi]\!](\overline{\mathrm{v}}) & = \bigcup_{v_x \in [\![\mathrm{Val}]\!]} ([\![\phi]\!](v_x, \overline{\mathrm{v}}))
\end{array}
$$

Now it is easy to verify by structural induction on formulas ϕ that the interpretation given in the BI hyperdoctrine S corresponds exactly to the forcing semantics given earlier:

Theorem 8. *$h \in [\![\phi]\!](v_1, \ldots, v_n)$ iff $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n], h \Vdash \phi$.*

As a consequence, we of course obtain the well-known result that separation logic is sound for classical first-order BI. But, more interestingly, the correspondence also shows that we may easily extend separation logic to higher-order since the BI hyperdoctrine S soundly models higher-order BI. We expand on this in the next section, which also discusses other consequences of the above correspondence. First, however, we explain that one can also obtain such a correspondence for other versions of separation logic.

An Intuitionistic Model. Consider again the set of heaps $(H_\bot, *)$ with an added bottom $\bot$, as above. We now define the order by

$$h_1 \sqsupseteq h_2 \qquad \text{iff} \qquad \mathrm{dom}(h_1) \subseteq \mathrm{dom}(h_2) \text{ and for all } x \in \mathrm{dom}(h_1).\ h_1(x) = h_2(x).$$

Let I be the set of sieves on H, i.e., downwards closed subsets of H, ordered by inclusion. This is a complete BI algebra, as can be verified directly or by an abstract argument [1, 14].

Now let T be the BI hyperdoctrine induced by the complete BI algebra I as in Example 6. The interpretation of predicate BI in this BI hyperdoctrine corresponds exactly to the intuitionistic pointer model of separation logic, which is presented using a forcing style semantics in [6].

The Permissions Model. It is also possible to fit the permissions model of separation logic from [4] into the framework presented here. The main point is that the set of heaps, which in that model map locations to values and permissions, has a binary operation $*$, which makes $(H_\bot, *)$ a partially ordered commutative monoid.

Remark 9. *The correspondences between separation logic and BI hyperdoctrines given above illustrate that what matters for the interpretation of separation logic is the choice of BI algebra. Indeed, the main relevance of the topos-theoretic constructions in [14] for models of separation logic is that they can be used to construct suitable BI-algebras (as subobject lattices in categories of sheaves).*

4 Consequences for Separation Logic

We have shown above that it is completely natural and straightforward to interpret first-order predicate BI in first-order BI-hyperdoctrines and that the standard pointer model of separation logic corresponds to a particular case of BI-hyperdoctrine. Based on this correspondence, in this section we draw some further consequences for separation logic.

4.1 Formalizing Separation Logic

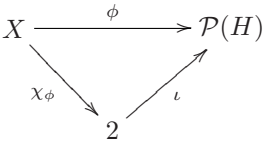
The strength of separation logic has been demonstrated in numerous papers before. In the early days of separation logic, it was shown that it could handle simple programs for copying trees, deleting lists, etc. The first proof of a more realistic program appeared in Yang's thesis [21], in which he showed correctness of the Schorr-Waite graph marking algorithm. Later, a proof of correctness of Cheney's garbage collection algorithm was published in [2], and other examples of correctness proofs of non-trivial algorithms may be found in [3]. In all of these papers, different simple extensions of core separation logic were used. For example, Yang used lists and binary trees as parts of his term language, and Birkedal et. al. introduced expression forms for finite sets and relations. It would seem that it is a weakness of separation logic that one has to come up with suitable extensions of it every time one has to prove a new program correct. In particular, it would make machine-verifiable formalizations of such proofs more burdensome and dubious if one would have to alter the underlying logic for every new proof.

We argue that the right way to look at these "extensions" is that they are really trivial definitional extensions of one and the same logic, namely the internal logic of the classical BI hyperdoctrine S presented in Section 3. The internal language of a BI hyperdoctrine $\mathcal{P}$ over $\mathcal{C}$ is formed as follows: to each object of $\mathcal{C}$ one associates a type, to each morphism of $\mathcal{C}$ one associates a function symbol, and to each predicate in $\mathcal{P}(X)$ one associates a relation symbol. The terms and formulas over this signature (considered as a higher-order signature [7]) form the internal language of the BI hyperdoctrine. There is an obvious structure for this language in $\mathcal{P}$.

Let $2 = \{\bot, \top\}$ be a two-element set (the subobject classifier of **Set**). There is a canonical map $\iota : 2 \to \mathcal{P}(H)$ that maps $\bot$ to $\{\}$ (the bottom element of the BI algebra $\mathcal{P}(H)$) and $\top$ to H (the top element of $\mathcal{P}(H)$).

Definition 10. *Let ϕ be an S-predicate over a set X, i.e., a function $\phi : X \to \mathcal{P}(H)$. Call ϕ* pure *if ϕ factors through ι.*

Thus $\phi : X \to \mathcal{P}(H)$ is pure if there exists a map $\chi_\phi : X \to 2$ such that



commutes. This corresponds to the notion of pure predicate traditionally used in separation logic [20].

The sub-logic of pure predicates is simply the standard classical higher-order logic of **Set**, and thus it is sound for classical higher-order logic. Hence one can use classical higher-order logic for defining lists, trees, finite sets and relations in the standard manner using pure predicates and prove the standard properties of these structures, as needed for the proofs presented in the papers referred to above. In particular, notice that recursive definitions of predicates, which in [21, 2, 3] are defined at the meta level, can be defined inside the higher-order logic itself. For machine verification one would thus only need to formalize one and the same logic, namely a sufficient fragment of the internal logic of the BI hyperdoctrine (with obvious syntactic rules for when a formula is pure). The internal logic itself is "too big" (it can have class-many types and function symbols, e.g.); hence the need for a fragment thereof, say classical higher-order logic with natural numbers.

4.2 Higher-Order Separation Logic

As mentioned in Section 3, the interpretation of separation logic in BI hyperdoctrines shows that we may extend separation logic to higher-order. Specifically, we may quantify over *any* set X in $\forall x : X.\phi$ and $\exists x : X.\phi$, including "pure sets" of trees, lists, etc., but also including propositions — the BI algebra $\mathcal{P}(H)$ — and predicates — sets of the form $(\mathcal{P}(H))^Y$, for some set Y. The quantification over "pure sets" has been usefully applied in program proving with separation logic, as mentioned in the previous section (it has also been usefully applied in Hoare logic, as pointed out to us by John Reynolds, see [18]). It remains to be seen to what extent quantification over general propositions and predicates is useful in actual program proving. But let us consider a simple example, which indicates that it may be useful. Consider the assertion $\exists P : \mathsf{prop}\,.\ P \mathrel{-\!\!*} Q$. Intuitively, is says that for *some* extension of the current heap, described by P, the combined heap will satisfy Q. Consider a canonical algorithm for copying a

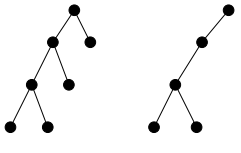


Fig. 1. Copying a tree

tree. To describe the invariant, we look at the snapshot in Fig. 1. Suppose the predicate $\mathsf{tree}\ \tau$ asserts that "the tree on the left in Fig. 1 is represented in the heap" (we are a bit informal here, but a formal presentation would clutter the point). Then the following assertion describes the situation in Fig. 1:

$$\mathsf{tree}\ \tau * ((\exists l_1, l_2, v_1, v_2.\ l_1 \mapsto v_1 * l_2 \mapsto v_2) \mathrel{-\!\!*} \mathsf{tree}\ \tau).$$

However, we might not care about the number of actual values and locations that are missing in the "new" tree in the heap, but just wish to express that some of the original tree has been copied, and that the original tree has not been manipulated. This can be done with the assertion

$$\mathsf{tree}\ \tau * (\exists P : \mathsf{prop}.\ P \mathrel{-\!\!*} \mathsf{tree}\ \tau).$$

Future research will show how useful *higher-order* separation logic is in actual proofs of programs.

4.3 Logical Characterizations of Classes of Assertions

Different classes of assertions, precise, monotone, and pure, were introduced in [20], and it was noted that special axioms for these classes of assertions are valid. Such special axioms were further exploited in [2], where pure assertions were moved in and out of the scope of iterated separating conjunctions, and in [11], where precise assertions were crucially used to verify soundness of the hypothetical frame rule. The different classes of assertions were defined semantically and the special axioms were also validated using the semantics. We now show how the higher-order features of higher-order separation logic may be used to logically characterize the classes of assertions and logically prove the properties earlier taken as axioms. This is, of course, important for machine verification, since it means that the special classes of assertions and their properties can be expressed *in the logic*.

To simplify notation we just present the characterizations for *closed* assertions, the extension to open assertions is straightforward. Recall that closed assertions are interpreted in S as functions from 1 to $\mathcal{P}(H)$, i.e., as subsets of H.

In the proofs below, we use assertions which describe heaps in a canonical way. Since a heap h has finite domain, there is a unique (up to permutation) way to write an assertion $p_h \equiv l_1 \mapsto n_1 * \ldots * l_k \mapsto n_k$ such that $[\![p_h]\!] = \{h\}$.

Precise Assertions. The traditional definition of a precise assertion is semantic, in that an assertion q is precise if, and only if, for all states s, h, there is at most one subheap h_0 of h such that $s, h_0 \Vdash q$. The following proposition logically characterizes closed precise assertions (at the semantic level, this characterization of precise predicates was mentioned in [10]).

Proposition 11. *The closed assertion q is precise if, and only if, the assertion*

$$\forall p_1, p_2 : \mathsf{prop}\,.\ (p_1 * q) \wedge (p_2 * q) \rightarrow (p_1 \wedge p_2) * q \tag{1}$$

is valid in the BI hyperdoctrine S.

Proof: The "only-if" direction is trivial, so we focus on the other implication. Thus suppose (1) holds for q, and let h be a heap with two different subheaps h_1, h_2 for which $h_i \in [\![q]\!]$. Let p_1, p_2 be canonical assertions that describe the heaps $h \setminus h_1$ and $h \setminus h_2$, respectively. Then $h \in [\![(p_1 * q) \wedge (p_2 * p)]\!]$, so $h \in [\![(p_1 \wedge p_2) * q]\!]$, whence there is a subheap $h' \subseteq h$ with $h' \in [\![p_1 \wedge p_2]\!]$. This is a contradiction. □

One can verify properties that hold for precise assertions *in the logic* without using semantical arguments. For example, one can show that $q_1 * q_2$ is precise if q_1 and q_2 are by the following logical argument: Suppose (1) holds for q_1, q_2. Then,

$$\begin{array}{ll} & (p_1 * (q_1 * q_2)) \wedge (p_2 * (q_1 * q_2)) \Rightarrow ((p_1 * q_1) * q_2) \wedge ((p_2 * q_1) * q_2)) \\ \Rightarrow & ((p_1 * q_1) \wedge (p_2 * q_1)) * q_2 \qquad \Rightarrow ((p_1 \wedge p_2) * q_1) * q_2 \\ \Rightarrow & (p_1 \wedge p_2) * (q_1 * q_2), \end{array}$$

as desired.

Monotone Assertions. A closed assertion q is defined to be *monotone* if, and only if, whenever $h \in [\![q]\!]$ then also $h' \in [\![q]\!]$, for all extensions $h' \supseteq h$.

Proposition 12. *The closed assertion q is* monotone *if, and only if, the assertion $\forall p : \mathsf{prop}\,.\; p * q \rightarrow q$ is valid in the BI hyperdoctrine S.*

This is also easy to verify, and again, one can show the usual rules for monotone assertions in the logic (without semantical arguments) using this characterization.

Pure Assertions. Recall from above that an assertion q is pure iff its interpretation factors through 2. Thus a closed assertion is pure iff its interpretation is either $\emptyset$ or H.

Proposition 13. *The closed assertion q is pure if, and only if, the assertion*

$$\forall p_1, p_2 : \mathsf{prop}\,.\; (q \wedge p_1) * p_2 \leftrightarrow q \wedge (p_1 * p_2) \tag{2}$$

is valid in the BI hyperdoctrine S.

Proof: Again, the interesting direction here is the "if" implication. Hence, suppose (2) holds for the assertion q, and that $h \in [\![q]\!]$. For any heap h_0, we must then show that $h_0 \in [\![q]\!]$. This is done via the verification of two claims.

Fact 1: For all $h' \subseteq h$, $h' \in [\![q]\!]$. Proof: Let p_1 be a canonical description of h', and p_2 a canonical description of $h \setminus h'$. Then $h \in [\![q \wedge (p_1 * p_2)]\!]$, so $h \in [\![(q \wedge p_1) * p_2]\!]$. This means that there is a split $h_1 * h_2 = h$ with $h_1 \in [\![q \wedge p_1]\!]$ and $h_2 \in [\![p_2]\!]$. But then, $h_2 = h \setminus h'$, so $h_1 = h'$, and thus, $h' \in [\![q]\!]$.

Fact 2: For all $h' \supseteq h$, $h' \in [\![q]\!]$. Proof: Let p_1 and p_2 be canonical descriptions of h and $h' \setminus h$, respectively. Then, $h' \in [\![(q \wedge p_1) * p_2]\!]$, so $h' \in [\![q \wedge (p_1 * p_2)]\!]$, and in particular, $h' \in [\![q]\!]$, as desired.

Using Facts 1 and 2, we deduce $h \in [\![q]\!] \;\Rightarrow\; \emptyset \in [\![q]\!] \;\Rightarrow\; h_0 \in [\![q]\!]$. □

4.4 Separation Logic for Richer Languages

Separation logic has mostly been used for low-level languages with a simple set-theoretic operational semantics. Yang [21–Ch. 9] has made some initial work on separation logic for richer languages such as Idealized Algol with heaps. For such richer languages, the semantics is typically given using more involved semantic structures such as functor categories and domains. We emphasize that all the developments in the present paper easily generalize to such more involved settings. Specifically, given any cartesian closed category $\mathcal{C}$ with an *internal* complete BI algebra B, one may construct a $\mathcal{C}$-indexed BI hyperdoctrine just as in Example 6.

5 Conclusion

We have introduced the notion of a (first-order) BI hyperdoctrine and shown that it soundly models classical and intuitionistic first- and higher-order predicate BI, thus connecting models of prediate BI with standard categorical notions of models of predicate logic. Moreover, we have shown that the standard pointer model of separation logic exactly corresponds to the interpretation of predicate BI in a BI hyperdoctrine. Finally, we have argued that this correspondence is of import for formalizations of separation logic, and that one can extend separation logic to higher-order.

Acknowledgements. The authors wish to thank Carsten Butz and the anonymous referees for their helpful comments and suggestions.

References

1. B. Biering. On the logic of bunched implications and its relation to separation logic. Master's thesis, University of Copenhagen, 2004.
2. L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. In *Proceedings of the 31-st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 220 – 231, Venice, Italy, 2004.
3. R. Bornat. Local reasoning, separation and aliasing. In *Proceedings of the Second Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'04)*, Venice, Italy, January 2004.
4. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Proceedings of POPL'05*, Long Beach, CA, USA, January 2005. ACM. Accepted for publication.
5. C. Calcagno, P. W. O'Hearn, and R. Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Computer Science*, 298(3):557 – 587, 2003.
6. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, 2001.
7. B. Jacobs. *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, 1999.

8. F.W. Lawvere. Adjointness in foundations. *Dialectica*, 23(3/4):281–296, 1969.
9. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, pages 1 – 19, Paris, France, September 2001. Springer Verlag.
10. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding (work in progress). Extended version of [11], 2003.
11. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proceedings of the 31-st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 268 – 280, Venice, Italy, 2004.
12. P.W. O'Hearn and D.J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 99.
13. A. M. Pitts. Categorical logic. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures*, chapter 2. Oxford University Press, 2000.
14. D. Pym, P. W. O'Hearn, and H. Yang. Possible worlds and resources: The semantics of BI. *Theoretical Computer Science*, 315(1):257 – 305, May 2004.
15. D. J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Kluwer Academic Publishers, 2002.
16. D. J. Pym. Errata and remarks for the semantics and proof theory of the logic of bunched implications. 2004. Available at `http://www.cs.bath.ac.uk/~pym/BI-monograph-errata.pdf`.
17. J. C. Reynolds. On extensions of separation logic. Private Communication.
18. J. C. Reynolds. *The Craft of Programming*. Prentice-Hall, 1981.
19. J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, Houndsmill, Hampshire, 2000.
20. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Seventeenth Annual IEEE symposium on Logic in Computer Science (LICS'02)*, pages 55 – 74, Copenhagen, Denmark, 2002.
21. H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois, Urbana-Champaign, 2001.
22. H. Yang and U. Reddy. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1):129 – 160, March 2004.

Deciding Reachability in Mobile Ambients

Nadia Busi and Gianluigi Zavattaro

Department of Computer Science, University of Bologna,
Mura A.Zamboni 7, 40127 Bologna (Italy)
{busi, zavattar}@cs.unibo.it

Abstract. Mobile Ambients has been proposed by Cardelli and Gordon as a foundational calculus for mobile computing. Since its introduction, the computational strength as well as the decidability of properties have been investigated for several fragments and variants of the standard calculus. We tackle the problem of reachability and we characterize a public (i.e., restriction free) fragment for which it is decidable. This fragment is obtained by removing the `open` capability and restricting the use of replication to guarded processes. Quite surprisingly, this fragment has been shown to be Turing complete by Maffeis and Phillips.

1 Introduction

Mobile Ambients (MA) [5] is a well known formalism exploited to describe distributed and mobile systems in terms of *ambients*. An ambient is a named collection of active processes and nested sub-ambients. In the pure (i.e., without communication) version of MA only three mobility primitives are used to permit ambient and process interaction: `in` and `out` for ambient movement, and `open` to dissolve ambient boundaries.

Following the tradition of process calculi, Mobile Ambients and its dialects have been equipped with a rich variety of formal tools useful for reasoning about and verifying properties of systems specified with these calculi (see, e.g., [9, 4, 6]). Another line of research regards the analysis of the expressiveness of these calculi in order to investigate the boundary between redundant and necessary features as well as the decidability of properties. For example, the Turing completeness of several variants and fragments of Mobile Ambients is investigated in [8], while the decidability of process termination (i.e. the existence of a finite computation) is investigated for fragments of the pure version of Mobile Ambients in [2].

Besides termination, an even more interesting property is process *reachability*: the reachability problem consists of verifying whether a target process can be reached from a source process. As an example of the relevance of reachability analysis, consider the system

$$intruder[P] \mid firewall[Q]$$

where an *intruder* running the program P attacks a $firewall$ executing the program Q. It is relevant to check whether the system

$$firewall[\ intruder[P'] \mid Q']$$

can be reached, where the intruder has succeeded.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 248–262, 2005.

The unique work, to the best of our knowledge, devoted to the investigation of reachability in Mobile Ambients is by Boneva and Talbot [1]. They prove that reachability is undecidable even in a minimal fragment of pure Mobile Ambients in which both the restriction operator (used to limit the scope of ambient names) and the `open` capability are removed.

Let us consider the above example of the *intruder* and the *firewall*. Traditional reachability consists of checking whether the target process is reachable for some instantiated processes P' and Q'. In general, one may be interested in concentrating only on the structure of the target process (i.e. the intruder is inside the firewall) abstracting away from the specific programs that run inside the ambients (i.e. abstracting away from P' and Q'). Exploiting classical reachability one should universally quantify on every possible processes P' and Q'.

To solve this problem we introduce *spatial reachability* permitting to specify a class of target processes. This class is characterized by a common structure of ambient nesting and a minimal number of processes that should be hosted inside those ambients. As an example of the use of spatial reachability consider the system

$$trojan[virus|P]|notebook[Q]$$

in which a *trojan* containing a *virus* program, and running program P, attacks a notebook running the program Q. One may be interested in checking whether the process

$$notebook[\ trojan[virus|P'] \mid Q']$$

can be reached for any possible P' and Q' not containing ambients. Observe that *virus* is a program for which it is necessary to check the actual presence inside the ambient *trojan* in the target process (a *trojan* that does not contain a *virus* is not dangerous).

We investigate the decidability of (spatial) reachability for fragments of the public, i.e. restriction free, version of the ambient calculus. We focus our analysis on calculi without restriction in order to concentrate on ambient nesting as the unique way for structuring processes. The relevance of restriction, as a mechanism for organizing processes inside name scopes, has been deeply investigated in the context of other process calculi such as the π–calculus [10].

The fragment that we characterize does not contain the `open` capability and limits the use of replication to guarded processes only (e.g., $!n[]$ is not a valid process for this fragment). This decidability result is proved by reducing reachability of processes to reachability in Petri nets (and spatial reachability to coverability). We prove the minimality of this fragment by showing that reachability becomes undecidable when relaxing at least one of the two restrictions imposed on the fragment. The undecidability for the `open`-free fragment has been proved by Boneva and Talbot [1]. For the fragment with guarded replication, we show how to reduce the halting problem for Random Access Machines [15] (a well known Turing powerful formalism) to the (spatial) reachability problem.

2 Pure Public Mobile Ambients

Pure public mobile ambients, that we denote with pMA, corresponds to the restriction-free fragment of the version of Mobile Ambients without communication defined by Cardelli and Gordon in [5].

Definition 1. – **pMA** – *Let Name, ranged over by n, m, ..., be a denumerable set of ambient names. The terms of pMA are defined by the following grammar:*

$$\begin{array}{lcl} P & ::= & \mathbf{0} \mid M.P \mid n[P] \mid P|P \mid !P \\ M & ::= & \mathtt{in}\, n \mid \mathtt{out}\, n \mid \mathtt{open}\, n \end{array}$$

We use $\prod_k P$ to denote the parallel composition of k instances of the process P, while $\prod_{i\in 1\ldots k} P_k$ denotes the parallel composition of the indexed processes P_i.

The term $\mathbf{0}$ represents the inactive process (and it is usually omitted). $M.P$ is a process guarded by one of the three mobility primitives (already discussed in the Introduction): after the execution of the primitive the process behaves like P. The processes $M.P$ are referred to as *guarded processes* in the following. The term $n[P]$ denotes an ambient named n containing process P. A process may be also the parallel composition $P|P$ of two subprocesses. Finally, the replication operator $!P$ is used to put in parallel an unbounded number of instances of the process P.

The operational semantics is defined in terms of a structural congruence plus a reduction relation.

Definition 2. – **Structural congruence** – *The structural congruence $\equiv$ is the smallest congruence relation satisfying:*

$$\begin{array}{ll} P \mid \mathbf{0} \equiv P & P \mid Q \equiv Q \mid P \\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R & !P \equiv P \mid !P \end{array}$$

Definition 3. – **Reduction relation** – *The reduction relation is the smallest relation $\rightarrow$ satisfying the following axioms and rules:*

$$(1)\quad n[\mathtt{in}\, m.P \mid Q] \mid m[R] \;\rightarrow\; m[n[P \mid Q] \mid R]$$

$$(2)\quad m[n[\mathtt{out}\, m.P \mid Q] \mid R] \;\rightarrow\; n[P \mid Q] \mid m[R]$$

$$(3)\quad \mathtt{open}\, n.P \mid n[Q] \;\rightarrow\; P \mid Q$$

$$(4)\quad \frac{P \;\rightarrow\; Q}{P \mid R \;\rightarrow\; Q \mid R}$$

$$(5)\quad \frac{P \;\rightarrow\; Q}{n[P] \;\rightarrow\; n[Q]}$$

$$(6)\quad \frac{P' \equiv P \quad P \;\rightarrow\; Q \quad Q' \equiv Q}{P' \;\rightarrow\; Q'}$$

As usual, we use $\rightarrow^*$ *to denote the reflexive and transitive closure of* $\rightarrow$*. If* $P \rightarrow^* Q$ *we say that* Q *is a* derivative *of* P*. The reachability problem consists in checking, given two processes* P *and* Q*, whether* Q *is a derivative of* P*, i.e. checking if* $P \rightarrow^* Q$*.*

Axioms (1), (2) and (3) describe the semantics of the three primitives `in`, `out` and `open`, respectively. A process inside an ambient n can perform an $\texttt{in}\, m$ operation in presence of a sibling ambient m; if the operation is executed then the ambient n moves inside m. If inside an ambient m there is an ambient n with a process performing an $\texttt{out}\, m$ action, this results in moving the ambient n outside the ambient m. Finally, a process performing an $\texttt{open}\, n$ operation has the ability to remove the boundary of an ambient $n[Q]$ composed in parallel with it.

Rules (4) and (5) are the contextual rules that respectively indicate that a process can move also when it is in parallel with another process and when it is inside an ambient. Finally, rule (6) is used to ensure that two structurally congruent terms have the same reductions.

In the paper we consider three fragments of pMA; $\text{pMA}_{g!}$ and $\text{pMA}^{-\texttt{open}}$ for which we show that reachability is undecidable and $\text{pMA}_{g!}^{-\texttt{open}}$ for which it turns out to be decidable.

Definition 4.
pMA$_{g!}$ *permits only guarded replication, i.e. it restricts the application of the replication operator to guarded processes:*[1]

$$\begin{array}{lcl} P & ::= & \mathbf{0} \mid M.P \mid n[P] \mid P|P \mid !M.P \\ M & ::= & \texttt{in}\, n \mid \texttt{out}\, n \mid \texttt{open}\, n \end{array}$$

pMA$^{-\texttt{open}}$ *removes the* `open` *capability:*

$$\begin{array}{lcl} P & ::= & \mathbf{0} \mid M.P \mid n[P] \mid P|P \mid !P \\ M & ::= & \texttt{in}\, n \mid \texttt{out}\, n \end{array}$$

pMA$_{g!}^{-\texttt{open}}$ *combines the restrictions imposed by the previous fragments:*

$$\begin{array}{lcl} P & ::= & \mathbf{0} \mid M.P \mid n[P] \mid P|P \mid !M.P \\ M & ::= & \texttt{in}\, n \mid \texttt{out}\, n \end{array}$$

3 Deciding Reachability in $\text{pMA}_{g!}^{-\texttt{open}}$

In this Section we show that reachability is decidable in $\text{pMA}_{g!}^{-\texttt{open}}$. We reduce reachability on $\text{pMA}_{g!}^{-\texttt{open}}$ to reachability on Place/Transition Petri nets.

[1] The structural congruence for $\text{pMA}_{g!}$ is obtained by replacing the axiom for replication with $!M.P \equiv M.P \mid !M.P$, and the the congruence rule for the replication operator with the congruence rule for the operator of restricted replication.

As reachability is decidable on such class of Petri nets [13], we get the decidability result for reachability on $\text{pMA}_{g!}^{-\texttt{open}}$.

Another interesting property is spatial reachability. Given two processes, P and R, the spatial reachability problem roughly consists in checking if, starting from P, it is possible to reach a process R' "greater" than R, in the following sense: the ambients in R and R' have the same structure of ambient nesting, and the (sequential and replicated) active subprocesses inside an R ambient are a subset of the subprocesses inside the corresponding ambient in R'. The Petri net constructed for the solution of the reachability problem can be exploited to reduce the spatial reachability problem for $\text{pMA}_{g!}^{-\texttt{open}}$ processes to the coverability problem for Petri nets, which is a decidable problem [14].

We start recalling Place/Transition nets with unweigthed flow arcs (see, e.g., [14]).

Definition 5. *Given a set S, a* finite multiset *over S is a function $m : S \to \mathbb{N}$ such that the set $dom(m) = \{s \in S \mid m(s) \neq 0\}$ is finite. The set of all finite multisets over S, denoted by $\mathcal{M}_{fin}(S)$, is ranged over by m. We write $m \subseteq m'$ if $m(s) \leq m'(s)$ for all $s \in S$. With $\oplus$ and $\setminus$ we denote* multiset union *and* multiset difference*, respectively.*

Definition 6. *A P/T net is a pair (S, T) where S is the set of* places *and $T \subseteq \mathcal{M}_{fin}(S) \times \mathcal{M}_{fin}(S)$ is the set of* transitions.

Finite multisets over the set S of places are called markings. *Given a marking m and a place s, we say that the place s contains $m(s)$* tokens.

A P/T net is finite if both S and T are finite.

A P/T system is a triple $N = (S, T, m_0)$ where (S, T) is a P/T net and m_0 is the initial marking.

A transition $t = (c, p)$ is usually written in the form $c \to p$. A transition $t = (c, p)$ is enabled *at m if $c \subseteq m$. The execution of a transition t enabled at m produces the marking $m' = (m \setminus c) \oplus p$. This is written as $m \xrightarrow{t} m'$ or simply $m \to m'$ when the transition t is not relevant.*

We say that m' is reachable from *m if there exists σ such that $m \xrightarrow{\sigma} m'$.*

We say that m' covers *m if $m \subseteq m'$.*

Definition 7. *Let $N = (S, T, m_0)$ be a P/T system.*

The reachability problem for marking m consists of checking if $m_0 \to^* m$.

The coverability problem for marking m consists of checking if there exists m' such that $m_0 \to^* m'$ and m' covers m.

3.1 Reducing Reachability on Processes to Reachability on Petri Nets

Now we show that reachability on processes can be reduced to reachability on Petri nets; by decidability of reachability on Petri nets, we get the following:

Theorem 1. *Let P, R be $pMA_{g!}^{-\texttt{open}}$ processes. The reachability problem $P \to^* R$ is decidable.*

Given two processes P and R, we outline construction of a (finite) Petri system $Sys_{P,R}$ satisfying the following property: the check of $P \rightarrow^* R$ is equivalent to check reachability of a finite set of markings on $Sys_{P,R}$. Because of the lack of space, the technical details concerning the construction of the net, as well as the auxiliary results needed to prove Theorem 1 are omitted; they can be found in [3].

The intuition behind this result relies on a monotonicity property of $\text{pMA}_{g!}^{-\texttt{open}}$: because of the absence of the `open` capability, the number of "active" ambients in a process (i.e., ambients that are not guarded by any capability) cannot decrease during the computation. Moreover, as the applicability of replication is restricted to guarded processes, the number of "active" ambients in a set of structurally equivalent processes is finite (while this is not the case in , e.g., the pMA process $!n[0]$). Thanks to the property explained above, in order to check if R is reachable from P it is sufficient to take into account a subset of the derivatives of P: namely, the P-derivatives whose number of active ambients is not greater than the number of active ambients in R.

Unfortunately, this subset of P-derivatives is, in general, not finite, as the processes inside an ambient can grow unlimitedly. Consider, e.g., the process $P = m[!\texttt{in}\, n.\texttt{out}\, n.Q] \mid n[]$: it is easy to see that, for any k, $m[\prod_k Q \mid !\texttt{in}\, n.\texttt{out}\, n.Q] \mid n[]$ is a derivative of P.

On the other hand, we note that the set of *sequential* and *replicated* terms that can occur as subprocesses of (the derivatives of) a process P (namely, the subterms of kind $M.P$ or $!M.P$) is finite. The idea is to borrow a technique used to map (the public fragment of) a process algebra on Petri nets. A process P is decomposed in the (finite) multiset of its sequential subprocesses that appear at top-level (i.e., occur unguarded in P); this multiset is then considered as the marking of a Place/Transition Petri net. The execution of a computational step in a process will correspond to the firing (execution) of a transition in the corresponding net. Thus, we reduce the reachability problem for $\text{pMA}_{g!}^{-\texttt{open}}$ processes to reachability of a finite set of markings in a Place/Transition Petri net, which is a decidable problem. However, differently from what happens in process algebras, where processes can be faithfully represented by a multiset of subprocesses, $\text{pMA}_{g!}^{-\texttt{open}}$ processes have a tree-like structure that hardly fits in a flat model such as a multiset.

The solution is to consider $\text{pMA}_{g!}^{-\texttt{open}}$ processes as composed of two kinds of components; the tree-like structure of ambients and the family of multisets of sequential subterms contained in each ambient. As an example, consider the process

$$\texttt{in}\, n.P \mid m[\texttt{in}\, n.P \mid \texttt{out}\, n.Q \mid n[\mathbf{0}] \mid k[\mathbf{0}] \mid \texttt{in}\, n.P] \mid n[\texttt{in}\, n.P]$$

having the tree-like structure $m[n[] \mid k[]] \mid n[]$. Moreover, there is a multiset corresponding to each "node" of the tree: the multiset $\{\texttt{in}\, n.P\}$ is associated to the root, the same multiset is associated to the n-labelled son of the root, the multiset $\{\texttt{in}\, n.P, \texttt{in}\, n.P, \texttt{out}\, n.Q\}$ is associated to the n-labelled son of the m-labelled son of the root, and so on.

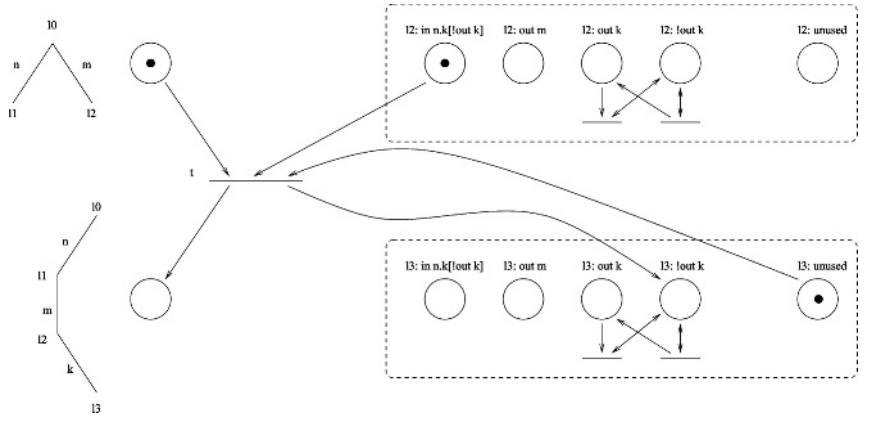


Fig. 1. A portion of the net corresponding to process $n[\texttt{out}\, m] \mid m[\texttt{in}\, n.k[!\texttt{out}\, k]]$

The Petri net we construct is composed of the following two parts: the first part is basically a finite state automaton, where the marked place represents the current tree-like structure of the process; the second part is a set of identical subnets: the marking of each subnet represents the multiset associated to a particular node of the tree. To keep the correspondence between the nodes of the tree and the multiset associated to that node, we make use of labels. A distinct label is associated to each subnet; this label will be used in the tree-like structure to label the node whose contents (i.e., the set of sequential subprocesses contained in the ambient corresponding to the node) is represented by the subnet.

The set of possible tree-like structures we need to consider is finite, for the following reasons. First of all, the set of ambient names in a process is finite. Moreover, to verify reachability we need to take into account only those processes whose number of active ambients is limited by the number of ambients in the process we want to reach.

The upper bound on the number of nodes in the tree-like structures also provides an upper bound to the number of identical subnets we need to decide reachability (at most one for each active ambient). In general, the number of active ambients grows during the computation; hence, we need a mechanism to remember which subnets are currently in use and which ones are not used. When a new ambient is created, a correspondence between the node representing such a new ambient in the tree-like structure and a not yet used subnet is established, and the places of the "fresh" subnet are filled with the marking corresponding to the sequential subprocesses contained in the newly created ambient. To this aim, each subnet is equipped with a place called *unused*, that contains a token as long as the subnet does not correspond to any node in the tree-like structure.

For example, consider the process $n[\texttt{out}\, m] \mid m[\texttt{in}\, n.k[!\texttt{out}\, k]]$. The relevant part of the net is depicted in Figure 1: a subset of the places, representing the tree-like structure, is depicted in the left-hand part of the figure, while the subnets are depicted in the right-hand part. We only report the subnets labelled

with l_2 and l_3, and omit the two subnets labelled with l_0 (with empty marking) and with l_1 (whose marking consists of a token in place $l_1 : \mathtt{out}\, m$). The computation step $n[\mathtt{out}\, m] \mid m[\mathtt{in}\, n.k[!\mathtt{out}\, k]] \rightarrow n[\mathtt{out}\, m \mid m[k[!\mathtt{out}\, k]]]$ corresponds to the firing of transition t in the net.

A last remark is concerned with structural congruence: because of the structural congruence rule (6), the reachability of a process R actually correspond to decide if it is possible to reach a process that is structurally congruent to R. As we are reducing the reachability in $\mathrm{pMA}_{g!}^{-\mathtt{open}}$ to marking reachability in Petri nets, it is necessary that the set of markings, corresponding to the set of processes structurally congruent to R, is finite. We concentrate on the markings of the subnets. The top-level applications of the monoidal laws for parallel composition are automatically dealt with, as processes that are structurally congruent because of such laws are mapped on the same marking. Unfortunately, the application of the replication law permits to produce an infinite set of markings corresponding to structurally congruent processes. Take, e.g., $!\mathtt{in}\, n.P \equiv \mathtt{in}\, n.P \mid !\mathtt{in}\, n.P \equiv \mathtt{in}\, n.P \mid \mathtt{in}\, n.P \mid !\mathtt{in}\, n.P \equiv \ldots$ and the corresponding set of markings $\{!\mathtt{in}\, n.P\}, \{\mathtt{in}\, n.P, !\mathtt{in}\, n.P\}, \{\mathtt{in}\, n.P, \mathtt{in}\, n.P, !\mathtt{in}\, n.P\} \ldots$.

To solve this problem, we make use of the following two techniques.

The top-level application of the law for replication can be easily dealt with by adding the transitions $!\mathtt{in}\, n.P \rightarrow !\mathtt{in}\, n.P \mid \mathtt{in}\, n.P$ and $!\mathtt{in}\, n.P \mid \mathtt{in}\, n.P \rightarrow !\mathtt{in}\, n.P$, respectively permitting to spawn a new copy of a replicated process and to absorbe a process that also appears in a replicated form in the marking. An instance of such transitions is depicted in the subnet $l2$ of Figure 1.

The last problem to be dealt with is the application of the laws in combination with the congruence law for prefix and ambient. Consider, e.g., the reachability of process $R = m[!\mathtt{in}\, n.!\mathtt{in}\, m.\mathbf{0}]$; concerning the subnet corresponding to the m-labelled son of the root, we must check reachability of an infinite set of markings, namely,

$$\{!\mathtt{in}\, n.!\mathtt{in}\, m.\mathbf{0}\}, \{!\mathtt{in}\, n.(\mathtt{in}\, m.\mathbf{0} \mid !\mathtt{in}\, m.\mathbf{0})\}, \{!\mathtt{in}\, n.(\mathtt{in}\, m.\mathbf{0} \mid \mathtt{in}\, m.\mathbf{0} \mid !\mathtt{in}\, m.\mathbf{0})\}, \ldots.$$

To this aim, we introduce canonical representations of the equivalence classes of structural congruence, roughly consisting in nested multisets where the presence of a replicated version of a sequential term forbids the presence of any occurrence of the nonreplicated version of the same term. For example, the normal form of process $\mathtt{in}\, n.(!\mathtt{out}\, m.\mathbf{0}) \mid !\mathtt{in}\, n.(\mathtt{out}\, m.\mathbf{0} \mid !\mathtt{out}\, m.\mathbf{0}) \mid n[\mathtt{in}\, n.\mathbf{0}]$ is the nested multiset $!\mathtt{in}\, n.(!\mathtt{out}\, m.\mathbf{0}) \mid n[\mathtt{in}\, n.\mathbf{0}]$.

Now we are ready to describe the net that will be used to decide reachability of a process R starting from a process P.

The set of places of the net is constructed as follows. The part of the net representing the tree-like structure contains a place for each tree of size not greater than the number of active ambients in R. Each of the subnets contains a place for each sequential and replicated subprocess of process P, and a place named "unused", that remains filled as long as the subnet does not correspond to any node in the tree-like structure. Moreover, we associate a distinct label to each subnet, and all the places of the subnet will be decorated with such a label.

The net has two sets of transitions: the first set permits to model the execution of the `in` and `out` capabilities, while the second set is used to cope with the structural congruence rule for replication.

We concentrate on the first set of transitions. A capability, say, e.g., $\texttt{in}\, n$, can be executed when the following conditions are fulfilled: the tree-like structure must have a specific structure and a place corresponding to a sequential subprocess $\texttt{in}\, n.Q$ is marked in a subnet whose label appears in the right position in the tree-like structure. Moreover, the number of active ambients created by the execution of the capability, added to the number of currently active ambients, must not exceed the number of active ambients in the process R we want to reach. This condition is checked by requiring that there exist a sufficient number of "unused" places that are currently marked. The execution of the capability causes the following changes to the marking of the net: the place corresponding to the new tree-like structure is now filled and the marking of the subnet performing the $\texttt{in}\, n$ operation is updated (by adding the tokens in the places corresponding to the active sequential and replicated subprocesses in the continuation Q). Moreover, a number of subnets equal to the number of active ambients in the continuation Q become active: their places will be filled with the tokens corresponding to the active sequential and replicated subprocesses contained in the corresponding ambient, and the tree-like structure is updated accordingly.

Besides deciding reachability, the net system described above can be used to check the weaker property of spatial reachability.

3.2 Spatial Reachability

The spatial reachability problem for processes P and R roughly consists in checking if, starting from P, it is possible to reach a process R' "greater than" R, in the following sense:

- R' has the same spatial ambient structure of R, and
- the sequential and replicated active subprocesses contained in each ambient of R are also present in the corresponding ambient of R'.

The $\preceq_s$ relation is a formalization of the "greater than" concept:

Definition 8. *Let P and Q be $pMA^{-\texttt{open}}_{g!}$ processes.*
$P \preceq_s Q$ iff

- *either $Q \equiv P \mid \prod_i M_i.P_i \mid \prod_j !M'_j.P'_j$,*
- *or $P \equiv P_1 \mid n[P_2]$, $Q \equiv Q_1 \mid n[Q_2]$ and $P_i \preceq_s Q_i$ for $i = 1, 2$*

The spatial reachability problem for processes P and R consists in checking if there exists R' such that $P \rightarrow^* R'$ and $R \preceq_s R'$.

The mapping of processes to Petri nets markings outlined above satisfies the following property: if $P_1 \preceq_s P_2$ then the marking corresponding to P_2 covers the marking corresponding to P_1. Hence, the Petri net constructed in the previous section permits to reduce the spatial reachability problem for processes P and R to a coverability problem. As coverability is decidable in P/T nets, we obtain the following:

Theorem 2. *Let P, R be $pMA^{-\texttt{open}}_{g!}$ processes. The spatial reachability problem for P and R is decidable.*

4 Undecidability Results

In this section we discuss the undecidability of reachability for the two fragments $\text{pMA}^{-\texttt{open}}$ and $\text{pMA}_{g!}$.

As far as $\text{pMA}^{-\texttt{open}}$ is concerned, we resort to an equivalent result proved by Boneva and Talbot in [1] for a slightly different calculus. That calculus, proposed in [6, 7], differs from $\text{pMA}^{-\texttt{open}}$ only for three extra rules in the definition of the structural congruence relation: $\mathbf{0} \equiv \mathbf{0}$, $!!P \equiv !P$, $!(P \mid Q) \equiv !P \mid !Q$. These rules are added by Boneva and Talbot to guarantee that the congruence relation is confluent, thus decidable.

The undecidability of reachability is proved by Boneva and Talbot showing how to encode two-counters machines [11], a well known Turing powerful formalism. The encoding preserves the *one-step property*: if the two-counters machine $2CM$ moves in one step to $2CM'$ then $[\![2CM]\!] \rightarrow^* [\![2CM']\!]$, where $[\![\,]\!]$ is the considered encoding. Even if the calculus in [1] is slightly different from $\text{pMA}^{-\texttt{open}}$, the encoding of two-counters presented in that paper applies also to our calculus; this because the encoding does not apply the replication operator to the empty process, to replicated processes and to parallel composition of processes (i.e. the cases in which the three extra structural congruence rules come into play, respectively).

As far as $\text{pMA}_{g!}$ is concerned, we present a modeling of Random Access Machines (RAMs) [15], a formalism similar to two-counters machines. The encoding that we present is an enhancement of the RAM modeling in [2]: the main novelties are concerned with a more restricted use of replication and a reshaping of the garbage processes.

4.1 Random Access Machines

RAMs are a computational model based on finite programs acting on a finite set of registers. More precisely, a RAM R is composed of the registers $r_1, \ldots, r_n$, that can hold arbitrary large natural numbers, and by a sequence of indexed instructions $(1 : I_1), \ldots, (m : I_m)$. In [12] it is shown that the following two instructions are sufficient to model every recursive function:

- $(i : \mathit{Succ}(r_j))$: adds 1 to the contents of register r_j and goes to the next instruction;
- $(i : \mathit{DecJump}(r_j, s))$: if the contents of the register r_j is not zero, then decreases it by 1 and goes to the next instruction, otherwise jumps to the instruction s.

The computation starts from the first instruction and it continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction number higher than the length of the program is reached. It is not restrictive to assume that the instruction number

reached at the end of the computation is always $m+1$, and to assume that the computation starts and terminates with all the registers empty.

4.2 Modelling RAMs in pMA$_{g!}$

We model instructions and registers independently. As far as the instructions and the program counter are concerned, we model the program counter i with an ambient $pc_i[]$. Each instruction I_i is represented with a replicated process guarded by the capability $\texttt{open}\, pc_i$ able to open the corresponding program counter ambient $pc_i[]$. The processes modeling the instructions are replicated because each instruction could be performed an unbounded amount of times during the computation.

The key idea underlying the modeling of the registers is to represent natural numbers with a corresponding nesting of ambients. We use an ambient named z_j to represent the register r_j when it is empty; when the register is incremented we move the ambient z_j inside an ambient named s_j, while on register decrement we dissolve the outer s_j ambient boundary. In this way, for instance, the resister r_j with content 2 is modeled by the nesting $s_j[s_j[z_j[]]]$ (plus other processes hosted in these ambients that are discussed in the following).

Definition 9. *Given the RAM R with instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$ we define $[\![R]\!]$ as the following process*

$$pc_1[] \mid \prod_{i\in 1\ldots m} !\texttt{open}\, pc_i.C_i \mid \prod_{j\in 1\ldots n} R_j^0 \mid \\ \texttt{open}\, pc_{m+1}.GC \mid !\texttt{open}\, msg \mid garbage[\texttt{open}\, gc]$$

where C_i (modeling the $i-th$ instruction), R_j^0 (modeling the empty register r_j) and GC (the garbage collector which is started at the end of the computation) are shorthand notations defined in the following.

Note the use of two extra processes: $!\texttt{open}\, msg$ used to open ambients containing messages produced during the computation and the ambient $garbage[\texttt{open}\, gc]$ which is a container for the produced garbage. The process $\texttt{open}\, gc$ is used at the end of the computation to allow the garbage collector to act inside the ambient *garbage* as detailed in the following.

The register r_j with content l is represented by the process R_j^l defined inductively as follows

$$\begin{aligned} R_j^0 \quad &= z_j[\, !\texttt{open}\, inc_j. \\ &\qquad (\ msg[\, \texttt{out}\, z_j.s_j[\ REG_j\ ]\ ] \mid \\ &\qquad\ \ \texttt{in}\, s_j.acki_j[\, \texttt{out}\, z_j.!\texttt{out}\, s_j]\ ) \mid \\ &\quad !\texttt{open}\, zero_j.ackz_j[\, \texttt{out}\, z_j.\texttt{in}\, dj_j\ ] \mid \\ &\quad \texttt{open}\, gc\ ] \\ R_j^{l+1} &= s_j[\ \ REG_j \mid R_j^l\ \ ] \end{aligned}$$

where REG_j is a shorthand notation defined as follows

$$REG_j = \texttt{open}\, dec_j.ackd_j[\, \texttt{out}\, s_j.in\ dj_j\ ] \mid !\texttt{open}\, msg$$

Also in this case, the process $\texttt{open}\, gc$ is used to allow the garbage collector to act inside the ambient z_j. We will discuss the behaviour of the term REG_j, and of the other processes inside the ambient z_j, after having discussed the encoding for the instructions.

Before formalizing the modeling of the instructions we anticipate that the names inc_j, $zero_j$ and dec_j are used to model requests for increment, test for zero and decrement of register r_j, respectively; the names $acki_j$, $ackz_j$ and $ackd_j$ models the corresponding acknowledgements produced by the registers to notify that a request has been managed.

The instructions are modeled as follows. If the i-th instruction is $Succ(r_j)$, its encoding is

$$\begin{aligned} C_i = {} & increq_j[\ !\texttt{in}\, s_j \mid \texttt{in}\, z_j.inc_j[\texttt{out}\, increq_j]\ ] \mid \\ & \texttt{open}\, acki_j.pc_{i+1}[] \end{aligned}$$

This modeling is based on two processes. The first one is the ambient $increq_j$ that represents a request for increment of the register r_j. The second process blocks waiting for an acknowledgement that will be produced after the actual increment of the register; when the acknowledgement is received, this process increments the program counter spawning $pc_{i+1}[]$.

The ambient $increq_j$ has the ability to enter the boundary of the ambient modelling the register r_j, to move through the nesting of ambients, and finally to enter the inner ambient z_j. After that, a new ambient inc_j exits the ambient $increq_j$ becoming in parallel with the processes of the ambient z_j. One of these processes (see the definition of R_j^0) detects the arrival of the new ambient and reacts by producing $s_j[REG_j]$; the ambient z_j then moves inside this new ambient. In this way the nesting of ambients s_j is incremented by one. After, the acknowledgement is produced in terms of an ambient named $acki_j$ that moves outside the register boundary.

If the i-th instruction is $DecJump(r_j, s)$ the encoding is as follows

$$\begin{aligned} C_i = {} & zero_j[\texttt{in}\, z_j] \mid dec_j[\texttt{in}\, s_j] \mid \\ & dj_j[\ ACKZ_{js} \mid ACKD_{ji}\ ] \end{aligned}$$

where

$$\begin{aligned} & ACKZ_{js} = \\ & \texttt{open}\, ackz_j.\texttt{in}\, garbage. \\ & \quad msg[\ \texttt{out}\, dj_j.\texttt{out}\, garbage.\texttt{open}\, dec_j.pc_s[]\ ] \\ & ACKD_{ji} = \\ & \texttt{open}\, ackd_j.\texttt{in}\, garbage. \\ & \quad msg[\ \texttt{out}\, dj_j.\texttt{out}\, garbage.\texttt{open}\, zeroj.\texttt{open}\, s_j.pc_{i+1}[]\ ] \end{aligned}$$

This modeling is based on three processes. The first process is an ambient named $zero_j$ which represents a request for a test for zero of the register r_j; the second process is an ambient named dec_j representing a request for decrement of the register r_j; the third process is an ambient named dj_j which is in charge to manage the acknowledgement produced by the register r_j. The acknowledgement indicates whether the decrement, or the test for zero request, has succeeded.

Let us consider the test for zero request. The ambient $zero_j[\texttt{in}\, z_j]$ has the ability to move inside the ambient z_j. This can occur only if the register r_j is currently empty; in fact, if r_j is not empty, the ambient z_j is not at the outer level. If the request enters the z_j ambient boundary, the processes inside the ambient z_j (see the definition of R_j^0) react by producing an acknowledgement modelled by an ambient named $ackz_j$ which moves inside the ambient dj_j.

Now, consider the request for decrement. The ambient $dec_j[\texttt{in}\, s_j]$ has the ability to enter the boundary of the process modelling the register r_j; this can occur only if the register is not empty (otherwise there is no ambient s_j). Inside the ambient s_j, the process REG_j reacts by producing an acknowledgement modelled by an ambient named $ackd_j$ which moves inside the ambient dj_j.

The processes inside the ambient dj_j have the ability to detect which kind of acknowledgement has been produced, and react accordingly. In case of $ackz_j$, the reaction is to move the ambient dj_j inside the ambient *garbage*, and to dissolve the boundary of the outer ambient dec_j. This is necessary to remove the decrement request that has failed. In case of $ackd_j$, the process also dissolves one of the boundaries s_j, in order to actually decrement the register. In both cases, the program counter is finally updated by either jumping to instruction s, or by activating the next instruction $i+1$, respectively.

This way of modeling RAMs does not guarantee the one-step preservation property because of the production of garbage, that is processes that are no more involved in the subsequent computation. More precisely, the following garbage is produced:

- each increment operation leaves an ambient $increq_j[!\texttt{in}\, s_j]$ inside the ambient z_j, plus the process $!\texttt{out}\, s_j$ at the outer level;
- each decrement operation leaves an ambient dj_j inside the ambient *garbage*, plus the two processes $\texttt{in}\, z_j$ and $!\texttt{open}\, msg$ at the outer level;
- each test for zero operation leaves an ambient dj_j inside the ambient *garbage*, plus the process $\texttt{in}\, s_j$ at the outer level.

Clearly, the exact shape of the garbage at the end of the modeling of the RAM computation is unpredictable because it depends on the exact number of instructions that are executed. Nevertheless, we use the garbage collector process GC, activated on program termination, in order to reshape the garbage in a predefined format.

The key idea underlying the garbage collection process is to exploit the structural congruence rule $!P \equiv P|!P$ used to unfold (and fold) replication. Consider an unpredictable amount of processes P in parallel, i.e. $\prod_n P$ with n unknown. If we add in parallel the process $!P$ we have that $\prod_n P \mid !P \equiv !P$, thus reshaping the process in a known format.

We are now in place for defining the garbage collector process formally

$$\begin{aligned} GC = {} & !!\texttt{out}\, s_j \mid !\texttt{in}\, z_j \mid !!\texttt{open}\, msg \mid !\texttt{in}\, s_j \mid \\ & \textstyle\prod_{j\in 1\ldots n}(gc[\ \texttt{in}\, z_j.(!\texttt{open}\, increq_j \mid !!\texttt{in}\, s_j)\ ]) \mid \\ & gc[\ \texttt{in}\, garbage \mid \\ & \quad \textstyle\prod_{j\in 1\ldots n}(\ !\texttt{open}\, dj_j \mid \prod_{i\in 1\ldots m}!ACKD_{ji} \mid \\ & \qquad \textstyle\prod_{s\in 1\ldots m+1}!ACKZ_{js}\ )\ ] \end{aligned}$$

The undecidability of reachability and spatial reachability is a trivial corollary of the following theorem. It is worth noting that in the statement of the theorem the register r_j (which is assumed to be empty at the end of the computation) is represented by a process which is the same as R_j^0 with the difference that the process $\texttt{open}\, gc$, initially available in the ambient z_j (see the definition of R_j^0), is replaced by the two processes $!\texttt{open}\, increq \mid !!\texttt{in}\, s_j$ left by the garbage collector.

Theorem 3. *Given the RAM R with instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$ we have that R terminates if and only if*

$$\begin{array}{l}
\prod_{i\in 1\ldots m} !\texttt{open}\, pc_i.C_i \mid \\
\prod_{j\in 1\ldots n}(\ z_j[\ !\texttt{open}\, inc_j. \\
\qquad\qquad (\ msg[\ \texttt{out}\, z_j.s_j[\ REG_j\ ]\ ]\ \mid \\
\qquad\qquad\ \ \texttt{in}\, s_j.acki_j[\ \texttt{out}\, z_j.!\texttt{out}\, s_j]\ )\ \mid \\
\qquad\qquad !\texttt{open}\, zero_j.ackz_j[\ \texttt{out}\, z_j.\texttt{in}\, dj_j\ ]\ \mid \\
\qquad\qquad !\texttt{open}\, increq_j \mid !!\texttt{in}\, s_j]\ )\ \mid \\
!!\texttt{out}\, s_j \mid !\texttt{in}\, z_j \mid !!\texttt{open}\, msg \mid !\texttt{in}\, s_j \mid \\
garbage[\ \prod_{j\in 1\ldots n}(\ !\texttt{open}\, dj_j \mid \prod_{i\in 1\ldots m} !ACKD_{ji} \mid \\
\qquad\qquad \prod_{s\in 1\ldots m+1} !ACKZ_{js}\ )\ ]
\end{array}$$

is reachable from the process $[\![R]\!]$ (as defined in Definition 9).

Moreover, we have that the RAM R terminates if and only if the process

$$pc_{m+1}[] \mid \prod_{j\in 1\ldots n} z_j[] \mid garbage[]$$

is spatially reachable from the process $[\![R]\!]$.

5 Conclusion

We have discussed the decidability of reachability in Mobile Ambients. We have characterized a fragment of the pure and public Mobile Ambients, namely the $\texttt{open}$-free fragment with guarded replication, for which reachability is decidable. We call this fragment $\mathrm{pMA}_{g!}^{-\texttt{open}}$. Our decidability result also holds for a variant of reachability, called spatial reachability, that permits to specify a class of target processes characterized by a common structure of ambient nesting.

The fragment $\mathrm{pMA}_{g!}^{-\texttt{open}}$ has been already investigated by Maffeis and Phillips in [8] (called L_{io} in that paper). They show that such a small fragment is indeed Turing complete, by providing an encoding of RAMs. The encoding they present permits to conclude that the existence of a terminating computation is an undecidable problem, while the decidability of reachability is raised as an open problem. Our decidability result provides a positive answer to this problem.

In order to prove the minimality of $\mathrm{pMA}_{g!}^{-\texttt{open}}$ we make use of (a slight adaptation of) the undecidability result by Boneva and Talbot [1]. They prove that reachability is undecidable for the $\texttt{open}$-free fragment, equipped with a structural congruence slightly different from the standard one (see the discussion in Section 4). Instead of getting decidability by imposing syntactical restrictions

(as we do for $\mathrm{pMA}_{g!}^{-\mathtt{open}}$), they move to a weaker version of the operational semantics. In particular, they show that reachability becomes decidable when the structural congruence law $!P \equiv P \mid !P$ is replaced by the reduction axiom $!P \rightarrow P \mid !P$.

Acknowledgements. We thank Jean-Marc Talbot and Iain Phillips for their insightful comments on a preliminary version of this paper.

References

1. I. Boneva and J.-M. Talbot. When Ambients Cannot be Opened. In *Proc. FOSSACS'03*, volume 2620 of *Lecture Notes in Computer Science*, pages 169-184. Springer-Verlag, Berlin, 2003. Full version to appear in Theoretical Computer Science, Elsevier.
2. N. Busi and G. Zavattaro. On the Expressive Power of Movement and Restriction in Pure Mobile Ambients. *in Theoretical Computer Science*, 322:477–515, Elsevier, 2004.
3. N. Busi and G. Zavattaro, Deciding Reachability in Mobile Ambients - Extended version. Available at `http://www.cs.unibo.it/ busi/papersaMA05.pdf/`
4. L. Cardelli, G. Ghelli, and A.D. Gordon. Types for the ambient calculus Information and Computation, 177(2):160–194, 2002.
5. L. Cardelli and A.D. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.
6. L. Cardelli and A.D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In Proc. of POPL'00, pages 365–377. ACM Press, 2000.
7. L. Cardelli and A.D. Gordon. Ambient Logic. Mathematical Structures in Computer Science, to appear.
8. S. Maffeis and I. Phillips. On the Computational Strength of Pure Mobile Ambients. To appear in Theoretical Computer Science, Elsevier, 2004.
9. M. Merro and M. Hennessy. Bisimulation congruences in safe ambients. In Proc. of POPL'02, pages 71–80, ACM Press, 2002.
10. R. Milner, J. Parrow, D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77. Academic Press, 1992.
11. M.L. Minsky. *Recursive Unsolvability of Post's Problem of "Tag" and others Topics in the Theory of Turing Machines. Annals of Math.*, 74:437–455, 1961.
12. M.L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, 1967.
13. C. Reutenauer. *The Mathematics of Petri Nets.* Masson, 1988.
14. W. Reisig. *Petri nets: An Introduction.* EATCS Monographs in Computer Science, Springer, 1985.
15. J.C. Shepherdson and J.E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.

Denotational Semantics for Abadi and Leino's Logic of Objects

Bernhard Reus and Jan Schwinghammer*

Informatics, University of Sussex, Brighton, UK
Fax +44 1273 877873
{bernhard, j.schwinghammer}@sussex.ac.uk

Abstract. Abadi-Leino Logic is a Hoare-calculus style logic for a simple imperative and object-based language where every object comes with its own method suite. Consequently, methods need to reside in the store ("higher-order store"). We present a new soundness proof for this logic using a denotational semantics where object specifications are recursive predicates on the domain of objects. Our semantics reveals which of the limitations of Abadi and Leino's logic are deliberate design decisions and which follow from the use of higher-order store. We discuss the implications for the development of other, more expressive, program logics.

1 Introduction

When Hoare presented his seminal work about an *axiomatic basis of computer programming* [7], high-level languages had just started to gain broader acceptance. Meanwhile programming languages are evolving ever more rapidly, whereas verification techniques seem to be struggling to keep up. For object-oriented languages several formal systems have been proposed, e.g. [2, 6, 14, 13, 5, 21, 20]. A "standard" comparable to the Hoare-calculus for imperative While-languages [4] has not yet emerged. Nearly all the approaches listed above are designed for class-based languages (usually a sub-language of sequential Java), where method code is known statically.

One notable exception is Abadi and Leino's work [2] where a logic for an object-based language is introduced that is derived from the imperative object calculus with first-order types, $\mathbf{imp}\varsigma$, [1]. In object-based languages, every object contains its own suite of methods. Operationally speaking, the store for such a language contains code (and is thus called *higher-order store*) and modularity is for free simply by the fact that all programs can depend on the objects' code in the store. We therefore consider object-based languages ideal for studying modularity issues that occur also in class-based languages. Class-based programs can be compiled into object-based ones (see [1]), and object-based languages can

* Supported by the EPSRC under grant GR/R65190/01, "Programming Logics for Denotations of Recursive Objects".

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 263–278, 2005.

naturally deal with classes defined on-the-fly, like inner classes and classes loaded at run-time (cf. [16, 17]).

Abadi and Leino's logic is a Hoare-style system, dealing with partial correctness of object expressions. Their idea was to enrich object types by method specifications, also called *transition relations*, relating pre- and post-execution states of program statements, and *result specifications* describing the result in case of program termination. Informally, an object satisfies such a specification

$$A \equiv [\mathsf{f}_i\colon {A_i}^{i=1\ldots n},\ \mathsf{m}_j\colon \varsigma(y_j)B_j :: {T_j}^{j=1\ldots m}]$$

if it has fields f_i satisfying A_i and methods m_j that satisfy the transition relation T_j and, in case of termination of the method invocation, their result satisfies B_j. However, just as a method can use the *self*-parameter, we can assume that an object a itself satisfies A in both B_j and T_j when establishing that A holds for a. This yields a powerful and convenient proof principle for objects.

We are going to present a new proof using a (untyped) denotational semantics (of the language and the logic) to define validity. Every program and every specification have a meaning, a *denotation*. Those of specifications are simply predicates on (the domain of) objects. The properties of these predicates provide a description of inherent limitations of the logic. Such an approach is not new, it has been used e.g. in LCF, a logic for functional programs [11].

The difficulty in this case is to establish predicates that provide the powerful reasoning principle for objects. Reus and Streicher have outlined in [19] how to use some classic domain theory [12] to guarantee existence and uniqueness of appropriate predicates on (isolated) objects. In an object-calculus program, however, an object may depend on other objects (and its methods) in the store. So object specifications must depend on specifications of other objects in the store which gives rise to "store specifications" (already present in the work of Abadi and Leino).

For the reasons given above, this paper is not "just" an application of the ideas in [19]. Much care is needed to establish the important invariance property of Abadi-Leino logic, namely that proved programs preserve store specifications. Our main achievement, in a nutshell, is that we have successfully applied the ideas of [19] to the logic of [2] to obtain a soundness proof that can be used to *analyse this logic* and to *develop similar but more powerful program logics* as well.

Our soundness proof is not just "yet another proof" either. We consider it complementary (if not superior) to the one in [2] which relies on the operational semantics of the object calculus and does not assign proper "meaning" to specifications. Our claim is backed up by the following reasons:

- By using denotational semantics we can introduce a clear notion of validity with no reference to derivability. This helps clarifying *what the proof is actually stating* in the first place.
- We can extend the logic easily e.g. for recursive specifications. This has been done for the Abadi-Leino logic in [9] but for a slightly different language with nominal subtyping.

– Some essential and unavoidable restrictions of the logic are revealed and justified.
– Analogously, it is revealed where restrictions have been made for the sake of simplicity that could be lifted to obtain a more powerful logic. For example, in [2] transition specifications cannot talk about methods at all.
– Our proof widens the audience for Abadi and Leino's work to semanticists and domain theorists.

The outline of this paper is as follows. In the next section, syntax and semantics of the object-calculus are presented. Section 3 introduces the Abadi-Leino logic and the denotational semantics of its object specifications. A discussion about store specifications and their semantics follows (Section 4). The main result is in Section 5 where the logic is proved sound. Of the various extensions discussed in Section 6, we present recursive specifications in some detail (Section 6.2). Section 7 concludes with a brief comparison to the original proof [2].

When presenting the language and logic, we deliberately keep close to the original presentation [2]. For a full version of this paper containing detailed proofs we refer to the technical report [18].

2 The Object Calculus

Below, we review the language of [2], which is based on the imperative object calculus of Abadi and Cardelli [1]. Following [19] we give a denotational semantics. The syntax of terms is defined by

$$\begin{aligned} a, b ::= {} & x \mid \texttt{true} \mid \texttt{false} \mid \texttt{if } x \texttt{ then } a \texttt{ else } b \mid \texttt{let } x = a \texttt{ in } b \\ & \mid [\mathsf{f}_i = {x_i}^{i=1\ldots n}, \mathsf{m}_j = \varsigma(y_j){b_j}^{j=1\ldots m}] \mid x.\mathsf{f} \mid x.\mathsf{f} := y \mid x.\mathsf{m} \end{aligned}$$

where $\mathsf{f} \in \mathcal{F}$ and $\mathsf{m} \in \mathcal{M}$ range over countably infinite sets of *field* and *method names*, resp. Object construction $[\mathsf{f}_i = x_i, \mathsf{m}_j = \varsigma(y_j)b_j]$ allocates new storage and returns (a reference to) an object containing fields f_i (with initial value the value of x_i) and methods m_j. In a method m_j, ς is a binder for the *self* parameter y_j in the method body b_j. During method invocation, the method body is evaluated with the self parameter bound to the host object.

We extend the syntax with integer constants and operations, and consider an object-based modelling of a bank account as an example:

$$\begin{aligned} acc(x) \equiv {} & \texttt{[balance = 0,} \\ & \texttt{deposit10 = } \varsigma(y) \texttt{ let } z \texttt{ = } y\texttt{.balance+10 in } y\texttt{.balance:=}z\texttt{,} \\ & \texttt{interest = } \varsigma(y) \texttt{ let } r \texttt{ = } x\texttt{.manager.rate in} \\ & \qquad\qquad \texttt{let } z \texttt{ = } y\texttt{.balance}*r\texttt{/100 in } y\texttt{.balance:=}z\texttt{]} \end{aligned}$$

Note how the self parameter y is used in both methods to access the `balance` field. Object *acc* depends on a "managing" object x in the context that provides the interest rate, through a field `manager`, for the `interest` method.

Semantics of Objects. We work in the category PreDom of predomains (cpos that do not necessarily contain a least element) and partial continuous functions. Let $A \rightharpoonup B$ denote the partial continuous function space between predomains A and B. For $f \in A \rightharpoonup B$ and $a \in A$ we write $f(a)\downarrow$ if f applied to a is defined, and $f(a)\uparrow$ otherwise.

If L is a set, then $\mathcal{P}(L)$ is its powerset, $\mathcal{P}_{\mathsf{fin}}(L)$ denotes the set of its finite subsets, and A^L is the set of all total functions from L to A. For a countable set $\mathbb{L}$ and a predomain A we write $\mathsf{Rec}_{\mathbb{L}}(A) = \sum_{L \in \mathcal{P}_{\mathsf{fin}}(\mathbb{L})} A^L$ for the predomain of *records* with entries from A and labels from $\mathbb{L}$. Note that $\mathsf{Rec}_{\mathbb{L}}$ extends to a locally continuous endofunctor on PreDom.

We write $\{\!|l_1 = a_1, \ldots, l_n = a_n|\!\}$ for a record $r = (L, f \in A^L)$, with labels $L = \{l_1, \ldots, l_n\}$ and corresponding entries $f(l_i) = a_i$. Update (and extension) $r[l := a]$ is defined in the obvious way. Selection of labels is written $r.l$.

The language of the previous section finds its interpretation within the following system of recursively defined predomains in PreDom:

$$\begin{aligned} \mathsf{Val} &= \mathsf{BVal} + \mathsf{Loc} & \mathsf{Ob} &= \mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}) \times \mathsf{Rec}_{\mathcal{M}}(\mathsf{Cl}) \\ \mathsf{St} &= \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Ob}) & \mathsf{Cl} &= \mathsf{St} \rightharpoonup (\mathsf{Val} + \{\mathsf{error}\}) \times \mathsf{St} \end{aligned} \tag{1}$$

where Loc is a countably infinite set of *locations* ranged over by l, and BVal is the of truth values *true* and *false*, considered as flat predomains.

Let $\mathsf{Env} = \mathsf{Var} \to_{\mathsf{fin}} \mathsf{Val}$ be the set of *environments*, i.e. maps between Var and Val with finite domain. Given an environment $\rho \in \mathsf{Env}$, the interpretation $[\![a]\!]\rho$ of an object expression a in $\mathsf{St} \rightharpoonup (\mathsf{Val} + \{\mathsf{error}\}) \times \mathsf{St}$ is given in Table 1, where the (strict) semantic **let** is also "strict" wrt. error. Note that for $o \in \mathsf{Ob}$ we just write $o.\mathsf{f}$ and $o.\mathsf{m}$ instead of $\pi_1(o).\mathsf{f}$ and $\pi_2(o).\mathsf{m}$, respectively. Similarly, we omit the injections for elements of $\mathsf{Val} + \{\mathsf{error}\}$. Because Loc is assumed to be infinite, the condition $l \notin \mathsf{dom}(\sigma)$ in the case for object creation can always be satisfied, i.e., object creation will never raise error because we run out of memory.

We will also use a projection to the part of the store that contains just data in Val (no closures), $\pi_{\mathrm{Val}} : \mathsf{St} \to \mathsf{St}_{\mathsf{Val}}$ defined by $(\pi_{\mathrm{Val}}\,\sigma).l.\mathsf{f} = \sigma.l.\mathsf{f}$, where $\mathsf{St}_{\mathsf{Val}} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}))$. We refer to $\pi_{\mathrm{Val}}(\sigma)$ as the *flat part* of σ.

3 Abadi-Leino Logic

We briefly recall Abadi and Leino's logic. For a more detailed presentation see [2, 8] or the technical report [18]. A *transition relation* T is a first-order formula over program variables that relates pre- and post-execution states of computations. There are function symbols $\grave{\sigma}$, $\acute{\sigma}$ and result that refer to the (flat parts of the) initial and final stores, and the result of a computation, resp. For instance, $\grave{\sigma}(x, \mathsf{f})$ denotes the value of $x.\mathsf{f}$ in the initial store, and analogously for $\acute{\sigma}$. Predicate symbols include T_{res}, T_{upd} and T_{obj} with the following meaning:

- $T_{\mathsf{res}}(x)$ holds if $\mathsf{result} = x$, and the (flat part of the) store remains unchanged
- $T_{\mathsf{upd}}(x, \mathsf{f}, y)$ holds if $\mathsf{result} = x$, $\acute{\sigma}(x, \mathsf{f}) = y$, and $\grave{\sigma}$ equals $\acute{\sigma}$ everywhere else
- $T_{\mathsf{obj}}(\mathsf{f}_1 = x_1, \ldots, \mathsf{f}_n = x_n)$ holds if result denotes a fresh location such that $x_i = \acute{\sigma}(\mathsf{result}, \mathsf{f}_i)$ for all i, and the store remains unchanged otherwise.

Table 1. Denotational semantics

$$[\![x]\!]\rho\sigma = \begin{cases} (\rho(x),\sigma) & \text{if } x \in \mathsf{dom}(\rho) \\ (\mathsf{error},\sigma) & \text{otherwise} \end{cases}$$

$$[\![\mathtt{true}]\!]\rho\sigma = (\mathit{true},\sigma)$$

$$[\![\mathtt{false}]\!]\rho\sigma = (\mathit{false},\sigma)$$

$$[\![\mathtt{if}\ x\ \mathtt{then}\ b_1\ \mathtt{else}\ b_2]\!]\rho\sigma = \begin{cases} [\![b_1]\!]\rho\sigma' & \text{if } [\![x]\!]\rho\sigma = (\mathit{true},\sigma') \\ [\![b_2]\!]\rho\sigma' & \text{if } [\![x]\!]\rho\sigma = (\mathit{false},\sigma') \\ (\mathsf{error},\sigma') & \text{if } [\![x]\!]\rho\sigma = (v,\sigma') \text{ for } v \notin \mathsf{BVal} \end{cases}$$

$$[\![\mathtt{let}\ x = a\ \mathtt{in}\ b]\!]\rho\sigma = \mathbf{let}\ (v,\sigma') = [\![a]\!]\rho\sigma\ \mathbf{in}\ [\![b]\!]\rho[x := v]\sigma'$$

$$[\![[\mathsf{f}_i = x_i{}^{i=1\ldots n}, \mathsf{m}_j = \varsigma(y_j)b_j{}^{j=1\ldots m}]]\!]\rho\sigma = \begin{cases} (l,\sigma[l := (o_1,o_2)]) & \text{if } x_i \in \mathsf{dom}(\rho), 1 \le i \le n \\ (\mathsf{error},\sigma) & \text{otherwise} \end{cases}$$

where $l \notin \mathsf{dom}(\sigma)$

$$o_1 = \{\!|\mathsf{f}_i = \rho(x_i)|\!\}^{i=1\ldots n}$$

$$o_2 = \{\!|\mathsf{m}_j = \lambda\sigma.[\![b_j]\!]\rho[y_j := l]\sigma|\!\}^{j=1\ldots m}$$

$$[\![x.\mathsf{f}]\!]\rho\sigma = \mathbf{let}\ (l,\sigma') = [\![x]\!]\rho\sigma\ \mathbf{in} \begin{cases} (\sigma'.l.\mathsf{f},\sigma') & \text{if } l \in \mathsf{dom}(\sigma') \text{ and } \mathsf{f} \in \mathsf{dom}(\sigma'.l) \\ (\mathsf{error},\sigma') & \text{otherwise} \end{cases}$$

$$[\![x.\mathsf{f} := y]\!]\rho\sigma = \mathbf{let}\ (l,\sigma') = [\![x]\!]\rho\sigma, (v,\sigma'') = [\![y]\!]\rho\sigma'\ \mathbf{in} \begin{cases} (l,\sigma''[l := \sigma''.l[\mathsf{f} := v]]) & \text{if } l \in \mathsf{dom}(\sigma'') \\ & \text{and } \mathsf{f} \in \mathsf{dom}(\sigma''.l) \\ (\mathsf{error},\sigma'') & \text{otherwise} \end{cases}$$

$$[\![x.\mathsf{m}]\!]\rho\sigma = \mathbf{let}\ (l,\sigma') = [\![x]\!]\rho\sigma\ \mathbf{in} \begin{cases} \sigma'.l.\mathsf{m}(\sigma') & \text{if } l \in \mathsf{dom}(\sigma') \text{ and } \mathsf{m} \in \mathsf{dom}(\sigma'.l) \\ (\mathsf{error},\sigma') & \text{otherwise} \end{cases}$$

Specifications combine transition relations for each method as well as the specifications of their results into a single specification for the whole object. They generalise the first-order types of [1], and are

$$A, B ::= \mathit{Bool} \mid [\mathsf{f}_i\colon A_i{}^{i=1\ldots n},\ \mathsf{m}_j\colon \varsigma(y_j)B_j :: T_j{}^{j=1\ldots m}]$$

where each T_j is a transition relation, and in general both B_j and T_j depend on the self parameter y_j.

Table 2 shows a specification for bank accounts as in the previous example.[1] Observe how the specification $T_{\mathtt{interest}}$ depends not only on the self parameter y of the host object but also on the statically enclosing object x.

Judgments of the logic are of the form $x_1{:}A_1, \ldots, x_n{:}A_n \vdash a : A :: T$. Informally, such a judgment means that if program a terminates when executed in a context where program variables $x_1, \ldots, x_n$ denote values that satisfy specifications $A_1, \ldots, A_n$, resp., then A describes properties of the result, and T describes the dynamic behaviour of a.

[1] Note that although we are using UML-like notation, these diagrams actually stand for individual objects, not classes – in fact there are no classes in the language.

Table 2. An example of transition and result specifications

$$T_{\texttt{deposit}}(y) \equiv \exists z.z = \grave{\sigma}(y, \texttt{balance}) \\ \wedge T_{\mathsf{upd}}(y, \texttt{balance}, z + 10)$$

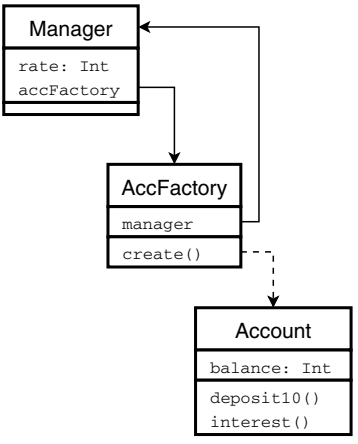
$$T_{\texttt{interest}}(x, y) \equiv \exists z.z = \grave{\sigma}(y, \texttt{balance}) \\ \wedge \exists m.m = \grave{\sigma}(x, \texttt{manager}) \\ \wedge \exists r.r = \grave{\sigma}(m, \texttt{rate}) \\ \wedge T_{\mathsf{upd}}(y, \texttt{balance}, z * r/100)$$

$$T_{\texttt{create}}(x) \equiv T_{\mathsf{obj}}(\texttt{balance} = 0)$$

$$A_{\texttt{Account}}(x) \equiv [\texttt{balance} : \mathit{Int}, \\ \texttt{deposit10} : \varsigma(y)[] :: T_{\texttt{deposit}}(y), \\ \texttt{interest} : \varsigma(y)[] :: T_{\texttt{interest}}(x, y)]$$

$$A_{\texttt{AccFactory}} \equiv [\texttt{manager} : [\texttt{rate} : \mathit{Int}], \\ \texttt{create} : \varsigma(x)A_{\texttt{Account}}(x) :: T_{\texttt{create}}(x)]$$

$$A_{\texttt{Manager}} \equiv [\texttt{rate} : \mathit{Int}, \\ \texttt{accFactory} : A_{\texttt{AccFactory}}]$$



We can use the proof rules of Abadi and Leino's logic to derive the judgment

$$x{:}A_{\texttt{AccFactory}} \vdash \mathit{acc}(x) : A_{\texttt{Account}}(x) :: T_{\mathsf{obj}}(\texttt{balance} = 0) \tag{2}$$

for the *acc* object. In the logic there is one rule for each syntactic form of the language. As indicated in the introduction, the most interesting and powerful rule of the logic is the object introduction rule,

$$\frac{\begin{array}{c} A \equiv [\mathsf{f}_i{:}\, {A_i}^{i=1\ldots n},\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}{T_j}^{j=1\ldots m}] \\ \Gamma \vdash x_i{:}A_i{::}T_{\mathsf{res}}(x_i)^{i=1\ldots n} \qquad \Gamma, y_j{:}A \vdash b_j{:}B_j{::}T_j^{j=1\ldots m} \end{array}}{\Gamma \vdash [\mathsf{f}_i = {x_i}^{i=1\ldots n}, \mathsf{m}_j = \varsigma(y_j){b_j}^{j=1\ldots m}] : A :: T_{\mathsf{obj}}(\mathsf{f}_i = {x_i}^{i=1\ldots n})}$$

In order to establish that the newly created object satisfies specification A one has to verify the methods b_j. When doing that one can *assume* that the host object (through the self parameter y_j) already satisfies A. Essentially, this causes the semantics of store specifications, introduced in the next section, to be defined by a mixed-variant recursion.

Using the object introduction rule, (2) can be reduced to a trivial proof obligation for the field `balance`, a judgment for the method `deposit10`,

$$\Gamma \vdash \texttt{let } z\texttt{=}(y\texttt{.balance})\texttt{+10 in } y\texttt{.balance:=}z : [] :: T_{\texttt{deposit}}(y) \tag{3}$$

where Γ is the context $x{:}A_{\texttt{AccFactory}}, y{:}A_{\texttt{Account}}(x)$, and a similar judgment for the method `interest`. A proof of (3) involves showing

$$\Gamma \vdash (y\texttt{.balance})\texttt{+10} : \mathit{Int} :: T_{\mathsf{res}}(\grave{\sigma}(y, \texttt{balance}) + 10) \tag{4}$$

$$\Gamma, z{:}\mathit{Int} \vdash y\texttt{.balance:=}z : [] :: T_{\mathsf{upd}}(y, \texttt{balance}, z) \tag{5}$$

for the constituents of the let expression. These can be proved from the rules for field selection and field update, resp., which have the general form

$$\frac{\Gamma \vdash x{:}[\mathsf{f}{:}A]{::}T_{\mathsf{res}}(x)}{\Gamma \vdash x.\mathsf{f}{:}A{::}T_{\mathsf{res}}(\check{\sigma}(x,\mathsf{f}))} \qquad \frac{\begin{array}{c}A \equiv [\mathsf{f}_i{:}\,A_i{}^{i=1\ldots n},\ \mathsf{m}_j{:}\,\varsigma(y_j)B_j{::}T_j{}^{j=1\ldots m}] \\ \Gamma \vdash x{:}A{::}T_{\mathsf{res}}(x) \quad \Gamma \vdash y{:}A_k{::}T_{\mathsf{res}}(y)\end{array}}{\Gamma \vdash x.\mathsf{f}_k := y{:}A{::}T_{\mathsf{upd}}(x,\mathsf{f}_k,y)}\ 1{\le}k{\le}n$$

The logic also provides a (structural) notion of subspecification, which generalises the usual notion of subtyping. So $\overline{x} \vdash A <: B$ holds if all fields of B are also fields of A with the same specification, and hereditarily all methods of B are methods of A with a stronger transition specification.

For instance, in the example in Tab. 2, $\vdash A_{\texttt{Manager}} <: [\texttt{rate} : \mathit{Int}]$ would be used in order to prove

$$m{:}A_{\texttt{Manager}}, x{:}A_{\texttt{AccFactory}} \vdash x.\texttt{manager:=}m : A_{\texttt{AccFactory}} :: T_{\mathsf{upd}}(x,\texttt{manager},m)$$

when creating the reference to the manager object in the `manager` field of the factory object.

Semantics of Specifications. We give a denotational semantics of specifications. Each transition relation $\overline{x} \vdash T$ with free variables contained in $\overline{x}$ denotes a predicate

$$[\![\overline{x} \vdash T]\!]\rho \in \mathcal{P}(\mathsf{St}_{\mathsf{Val}} \times \mathsf{Val} \times \mathsf{St}_{\mathsf{Val}})$$

depending on an environment ρ. This can be defined in a straightforward way [18]. Observe that the meaning of a transition relation $\vdash T$ without free variables does not depend on the environment, and we sometimes simply write $[\![T]\!]$ in this case.

Similarly, an object specification $\overline{x} \vdash A$ gives rise to a predicate that depends on values for the free variables (since the underlying logic in the transition relations is untyped, the specifications of the free variables $\overline{x}$ are not relevant here). The interpretation of specifications

$$[\![\overline{x} \vdash A]\!]\rho \in \mathcal{P}(\mathsf{Val} \times \mathsf{St})$$

is given in Table 3. Subspecifications are simply set containment: If $\overline{x} \vdash A <: B$ then $[\![\overline{x} \vdash A]\!]\rho \subseteq [\![\overline{x} \vdash B]\!]\rho$.

4 Store Specifications

Object specifications are not sufficient. This is a phenomenon of languages with higher-order store well known from subject reduction and type soundness proofs (see [1–Ch. 11], [10]). Since statements may call subprograms residing in the store it has to be verified as well.

The standard remedy – also used in [2] – is to relativise the typing judgement such that it only needs to hold for "verified" stores. In other words, judgements

Table 3. Semantics of specifications

$$[\![\overline{x} \vdash \mathit{Bool}]\!]\rho = \mathsf{BVal} \times \mathsf{St}$$

$$[\![\overline{x} \vdash [\mathsf{f}_i\colon A_i{}^{i=1\ldots n},\ \mathsf{m}_j\colon \varsigma(y_j)B_j{::}T_j{}^{j=1\ldots m}]]\!]\rho =$$

$$\left\{ (l,\sigma) \in \mathsf{Loc} \times \mathsf{St} \;\middle|\; \begin{array}{ll} \textbf{(F)} & \text{for all } 1 \le i \le n.\ \sigma.l.\mathsf{f}_i \in [\![\overline{x} \vdash A_i]\!]\rho \\ \textbf{(M)} & \text{for all } 1 \le j \le m, \text{ if } \sigma.l.\mathsf{m}_j(\sigma) = (v,\sigma')\downarrow \\ & \quad \text{then } (v,\sigma') \in [\![\overline{x}, y_j \vdash B_j]\!]\rho[y_j := l] \\ & \quad \text{and } (\pi_{\mathrm{Val}}(\sigma), v, \pi_{\mathrm{Val}}(\sigma')) \in [\![\overline{x}, y_j \vdash T_j]\!]\rho[y_j := l] \end{array} \right\}$$

are interpreted wrt. *store specifications*. A store specification assigns a specification to each location in a store. When an object is created, the specification assigned to it at the time of creation is included in the store specification.

In this section we will interpret such store specifications using the techniques from [19]. Since their denotations will rely on mixed-variant recursion, it is impossible to define a semantic notion of subspecification. Alas, the Abadi-Leino logic makes essential use of subspecifications. We get around this problem by only using a subset relationship on (denotations of) object specifications (where there is no contravariant occurrence of store as the semantics of objects is w.r.t. one fixed store, cf. Table 3).

Unfortunately, we are restricted by the logic's requirement that verified statements never break the validity of store specifications. In the case of field update this implies that subspecifications need to be invariant in their fields. As the semantic interpretation of the subspecification relation cannot reflect this, we were forced to sometimes use the syntactic subspecification relation.

Store Specifications and their Semantics. A store specification Σ assigns *closed* specifications to (a finite set of) locations:

Definition 1 (Store Specification). *A* store specification Σ *is a record* $\Sigma \in \mathsf{Rec}_{\mathsf{Loc}}(\mathit{Spec})$ *s.t.* $\Sigma.l = A$ *implies* $\vdash A$. *For store specifications* Σ, Σ' *we say* Σ' extends Σ, *written* $\Sigma' \succcurlyeq \Sigma$, *if* $\Sigma.l = \Sigma'.l$ *for all* $l \in \mathsf{dom}(\Sigma)$.

Because we focus on closed specifications in the following, we need a way to turn the components B_j of a specification $[\mathsf{f}_i\colon A_i{}^{i=1\ldots n},\ \mathsf{m}_j\colon \varsigma(y_j)B_j{::}T_j{}^{j=1\ldots m}]$ (recall that they may depend on y_j) into closed specifications. This is done by extending the syntax of expressions with locations: There is one symbol $\underline{l}$ for each $l \in \mathsf{Loc}$. When clear from context we will simply write l in place of $\underline{l}$. Further we write $A[\rho/\Gamma]$ for the simultaneous substitution of all $x \in [\Gamma]$ in A by $\underline{\rho(x)}$.

We can then abstract away from particular stores $\sigma \in \mathsf{St}$, and interpret closed result specifications $\vdash A$ with respect to such store specifications:

Definition 2 (Object Specifications). *For closed* A *let* $||A||_\Sigma \subseteq \mathsf{Val}$ *be*

$$||\mathit{Bool}||_\Sigma = \mathsf{BVal}$$
$$||A||_\Sigma = \{l \in \mathsf{Loc} \mid \vdash \Sigma.l <: A\}$$

where $A \equiv [\mathsf{f}_i{:}\, {A_i}^{i=1\ldots n},\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}{T_j}^{j=1\ldots m}]$. *This extends to contexts in the natural way.*

Observe that for all A, if $\Sigma' \succcurlyeq \Sigma$ then $||A||_\Sigma \subseteq ||A||_{\Sigma'}$. We obtain the following lemma about *context extensions*.

Lemma 1 (Context Extension). *If* $\rho \in ||\Gamma||_\Sigma$, $\Gamma, x{:}A$ *is a well-formed context and* $v \in ||A[\rho/\Gamma]||_\Sigma$ *then* $\rho[x := v] \in ||\Gamma, x{:}A||_\Sigma$.

In light of the object introduction rule, we would like to interpret store specifications as predicates over stores, as follows.

$\sigma \in [\![\Sigma]\!] :\Leftrightarrow$
  $\forall l \in \mathsf{dom}(\Sigma)$ where $\Sigma.l = [\mathsf{f}_i{:}\, {A_i}^{i=1\ldots n},\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}{T_j}^{j=1\ldots m}]$:
    (F) $\sigma.l.\mathsf{f}_i \in ||A_i||_\Sigma$ for all $1 \le i \le n$, and
    (M) $\forall \Sigma' \succcurlyeq \Sigma\ \forall \sigma' \in [\![\Sigma']\!]\ \forall v \in \mathsf{Val}\ \forall \sigma'' \in \mathsf{St}$, if $\sigma.l.\mathsf{m}_j(\sigma') = (v, \sigma'')\downarrow$ then
      (M1) $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![T_j[l/y_j]]\!]$
      (M2) $\exists \Sigma'' \succcurlyeq \Sigma'$ s.t. $\sigma'' \in [\![\Sigma'']\!]$
      (M3) $v \in ||B_j[l/y_j]||_{\Sigma''}$, for all $1 \le j \le m$

The universal quantification over extensions Σ' in **(M)** acccounts for (the specifications) of objects allocated between definition time and call time of methods. The existential quantification over extensions Σ'' in **(M2)** and **(M3)** provides for objects allocated by the method. In particular, since the result of a method call may be a freshly allocated object it is not sufficient to simply use Σ' in **(M2)** and **(M3)**. This semantic structure also appears in possible world models of other languages with dynamic allocation [10, 15].

Note the contravariant occurrence of $[\![-]\!]$ in $\forall \sigma' \in [\![\Sigma']\!]$ in **(M)**. Unfortunately, the usual techniques for establishing the existence of such predicates involving a mixed-variance recursion [12, 19] do not apply. They require the functional corresponding to the above recursion to map admissible predicates to admissible predicates. Due to the existential quantification in **(M2)** and **(M3)** this is not the case here.

We get around this problem by observing that the dynamic behaviour of programs (wrt. allocation of storage) can in fact be described more exactly, and the existential quantifier can be replaced: The elements of the (recursively defined) domain

$$\phi \in \mathsf{SF} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{M}}(\mathsf{St} \times \mathsf{SF} \times \mathit{Spec} \rightharpoonup \mathit{Spec} \times \mathsf{SF})) \tag{6}$$

are called *choice functions*, or *Skolem Functions*. The intuition is that, given a store $\sigma \in [\![\Sigma]\!]$, if $\sigma' \in [\![\Sigma']\!]$ with choice function ϕ', for some extension $\Sigma' \succcurlyeq \Sigma$ and the method invocation $\sigma.l.\mathsf{m}(\sigma')$ terminates, then $\phi.l.\mathsf{m}(\sigma', \phi', \Sigma') = (\Sigma'', \phi'')$ yields a store specification $\Sigma'' \succcurlyeq \Sigma'$ such that $\sigma'' \in [\![\Sigma'']\!]$ (and ϕ'' is a choice function for the extension Σ'' of Σ).

Using SF in the definition below has the effect of constraining the existential quantifier to work *uniformly* on the elements of increasing chains.

Definition 3 (Store Predicate). *Let* $\mathcal{P} = \mathcal{P}(\mathsf{St} \times \mathsf{SF})^{\mathsf{Rec}_{\mathsf{Loc}}(Spec)}$ *denote the collection of families of subsets of* $\mathsf{St} \times \mathsf{SF}$*, indexed by store specifications. We define a functional* $\Phi : \mathcal{P}^{op} \times \mathcal{P} \to \mathcal{P}$ *as follows.*

$(\sigma, \phi) \in \Phi(Y, X)_\Sigma :\Leftrightarrow$
(1) $\mathsf{dom}(\Sigma) = \mathsf{dom}(\phi)$ *and* $\forall l \in \mathsf{dom}(\Sigma).\, \mathsf{dom}(\pi_2(\Sigma.l)) = \mathsf{dom}(\phi.l)$*, and*
(2) $\forall l \in \mathsf{dom}(\Sigma)$ *where* $\Sigma.l = [\mathsf{f}_i\colon {A_i}^{i=1\ldots n},\ \mathsf{m}_j\colon \varsigma(y_j)B_j :: {T_j}^{j=1\ldots m}]$ *:*
(F) $\sigma.l.\mathsf{f}_i \in ||A_i||_\Sigma$ *for all* $1 \le i \le n$*, and*
(M) $\forall \Sigma' \succcurlyeq \Sigma\ \forall(\sigma', \phi') \in Y_{\Sigma'}.$ *if* $\sigma.l.\mathsf{m}_j(\sigma') = (v, \sigma'')\!\downarrow$ *then*
(M1) $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![T_j[l/y_j]]\!]$
(M2) $\phi.l.\mathsf{m}_j(\sigma', \phi', \Sigma') = (\Sigma'', \phi'')$ *s.t.* $\Sigma'' \succcurlyeq \Sigma'$ *and* $(\sigma'', \phi'') \in X_{\Sigma''}$
(M3) $v \in ||B_j[l/y_j]||_{\Sigma''}$ *for all* $1 \le j \le m$

We write $\sigma \in [\![\Sigma]\!]$ *if there is some* $\phi \in \mathsf{SF}$ *s.t.* $(\sigma, \phi) \in \mathit{fix}(\Phi)_\Sigma$*.*

Lemma 2. *Functional* Φ*, defined in Def. 3, does have a unique fixpoint.*

Proof. Firstly, one shows that Φ is monotonic and maps admissible predicates to admissible predicates, in the sense that for all X and Y,

$$\forall \Sigma.\ X_\Sigma \subseteq \mathsf{St} \times \mathsf{SF} \text{ admissible } \Rightarrow\ \forall \Sigma.\ \Phi(Y, X)_\Sigma \subseteq \mathsf{St} \times \mathsf{SF} \text{ admissible}$$

Next, define for all admissible $X, Y \in \mathcal{P}$, $e_1 \in [\mathsf{St} \rightharpoonup \mathsf{St}]$ and $e_2 \in [\mathsf{SF} \rightharpoonup \mathsf{SF}]$:

$$\langle e_1, e_2\rangle : X \subset Y \ \mathit{iff}\ \forall \Sigma, \sigma, \phi.\ (\sigma, \phi) \in X_\Sigma \wedge \langle e_1, e_2\rangle(\sigma, \phi)\!\downarrow \Rightarrow\ \langle e_1, e_2\rangle(\sigma, \phi) \in Y_\Sigma$$

such that $e : X \subset Y$ states that $e = \langle e_1, e_2\rangle$ maps pairs of stores and choice functions that are in X_Σ to pairs of stores and choice functions that are in corresponding component Y_Σ of Y. Let F be the locally continuous, mixed-variant functor associated with the domain equations (1) and (6), for which $F((\mathsf{St}, \mathsf{SF}), (\mathsf{St}, \mathsf{SF})) = (\mathsf{St}, \mathsf{SF})$ is the minimal invariant [12]. According to [12] it only remains to be shown that

$$e : X \subset X' \ \wedge\ e : Y' \subset Y \ \Rightarrow\ F(e, e) : \Phi(Y, X) \subset \Phi(Y', X') \qquad (\dagger)$$

for all $X, Y, X', Y' \in \mathcal{P}$ and $e \sqsubseteq \mathit{id}_{\mathsf{St}\times\mathsf{SF}}$ which follows from a similar line of reasoning as in [19]

Predicates denoting transition specifications must be upward-closed in the pre-execution store and downward-closed in the post-execution store. This holds in Abadi-Leino logic as transition specifications are only defined on the flat part of the store; if they referred to the method part, (†) could not necessarily be shown.

The next lemma establishes the relation between store and object specifications.

Lemma 3. *For all object specifications* A*, store specifications* Σ*, stores* σ*, and locations* l*, if* $\sigma \in [\![\Sigma]\!]$ *and* $l \in \mathsf{dom}(\Sigma)$ *such that* $\vdash \Sigma.l <: A$ *then* $(l, \sigma) \in [\![A]\!]$*.*

5 Soundness

We can now define the semantics of judgements of Abadi-Leino logic and prove the key lemma.

Definition 4 (Validity). *$\Gamma \vDash a : A :: T$ if and only if for all store specifications $\Sigma \in \mathsf{Rec}_{\mathsf{Loc}}(Spec)$, for all $\rho \in ||\Gamma||_\Sigma$ and all $\sigma \in [\![\Sigma]\!]$, if $[\![a]\!]\rho\sigma = (v, \sigma')$ then $(v, \sigma') \in [\![[\Gamma] \vdash A]\!]\rho$ and $(\pi_{\mathrm{Val}}(\sigma), v, \pi_{\mathrm{Val}}(\sigma')) \in [\![[\Gamma] \vdash T]\!]\rho$.*

Lemma 4 (Soundness and Invariance). *Suppose*
(H1) *$\Gamma \vdash a : A :: T$*
(H2) *$\Sigma \in \mathsf{Rec}_{\mathsf{Loc}}(Spec)$ is a store specification*
(H3) *$\rho \in ||\Gamma||_\Sigma$*

Then there exists $\phi \in [\mathsf{St} \times \mathsf{SF} \times Spec \rightharpoonup Spec \times \mathsf{SF}]$ s.t. for all $\Sigma' \succcurlyeq \Sigma$ and for all $(\sigma', \phi') \in fix(\Phi)_{\Sigma'}$, if $[\![a]\!]\rho\sigma' = (v, \sigma'')\downarrow$ then the following holds:
(S1) *there exists $\Sigma'' \succcurlyeq \Sigma'$ and $\phi'' \in \mathsf{SF}$ s.t. $\phi(\sigma', \phi', \Sigma') = (\Sigma'', \phi'')$*
(S2) *$(\sigma'', \phi'') \in fix(\Phi)_{\Sigma''}$*
(S3) *$v \in ||A[\rho/\Gamma]||_{\Sigma''}$*
(S4) *$(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![[\Gamma] \vdash T]\!]\rho$*

Note that condition **(S1)** explicates that store specifications are preserved by the execution of proved programs, which allows the inductive proof to go through.

Proof. The proof is by induction on the derivation of $\Gamma \vdash a : A :: T$ (whereas the original proof [2] is by induction on the length of the computation). Generally, we have to distinguish the case of objects, which are stored in the heap, and Booleans, which are stack allocated.

- Lemma 1 is applied in the cases (let) and (object construction), where an extended specification context is used in the induction hypothesis.
- Invariance of the field components in subspecifications is needed in the case for (field update).
- In the cases where the store changes, i.e., (object construction) and (field update), we must show explicitly that the resulting store satisfies the store specification, according to Definition 3. This is tedious but not difficult, due to the definition of $\sigma \in [\![\Sigma]\!]$.

Lemma 3 and Lemma 4 immediately prove

Theorem 1 (Soundness). *If $\Gamma \vdash a : A :: T$ then $\Gamma \vDash a : A :: T$.*

6 Denotational Analysis of Abadi-Leino Logic

For the proof of Theorem 2, establishing the existence of store predicates, it is necessary that transition relations are upwards and downwards closed in their first and second store argument, respectively. A sufficient condition is that transition relations work on the flat part of stores only. This provides an explanation why the transition relations of the Abadi-Leino do not refer to methods.

6.1 Extensions

This section contains a list of possible extensions of the Abadi-Leino logic. We think that the denotational semantics helps to clarify their feasibility.

Invariants of Fields. Abadi and Leino's logic is peculiar in that verified programs need to preserve store specifications. Put differently, only properties which are in fact preserved can be expressed in object specifications. In particular, specifying fields in object specifications is limited. Invariants like e.g. $\texttt{balance} \geq 0$, stating that an account comes without overdraft, cannot be formulated. The same axiom in a transition specification would only guarantee that the actual balance is positive. For "private" (local) fields, invisible to other objects, such invariants can be easily accommodated.

Method Parameters. Formal method parameters of the form $x : A$ can be attached to method specifications, e.g.,

$$\texttt{deposit}(x : Int) : \varsigma(y)[] :: \exists z.z = \grave{\sigma}(y, \texttt{balance}) \wedge T_{\mathsf{upd}}(y, \texttt{balance}, z + x)$$

by adding an extra assumption to the definition of store specifications. When $\sigma' \in [\![\Sigma']\!]$ then **(M1)**–**(M3)** have to be shown for all $v \in ||A||_{\Sigma'}$ where v is the actual parameter replacing formal parameter x in the method call.

Dynamic Loading. Dynamic loading of objects is, in a way, already available in the object calculus (this is one of its advantages over class-based languages). Loading an object of which only its specification $A \equiv [\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}\varsigma(x_j)B_j{::}T_j]$ is known corresponds to using a command of which one only knows its result specification A. Thus, $x : [m : \varsigma(y)A :: \exists \overline{z}.\ T_{\mathsf{obj}}(\mathsf{f}_i = z_i)] \vdash x.m() : A :: \exists \overline{z}.\ T_{\mathsf{obj}}(\mathsf{f}_i = z_i)$ describes dynamic loading where the load command is $x.m()$. It can be used to load any object fulfilling specification A.

Parametric Method Specifications. Transition specifications cannot refer to methods. While this is adequate when all method specifications are known it prevents verification of programs that use delegations (similar to the Command pattern). The flatness of transition relations is sufficient but not necessary for the existence of store specifications. Therefore "parametric" method specifications may be possible.

Method Update. Although method update is not allowed in Abadi-Leino logic, fields can be updated and thus the methods in a field object (similar to the Decorator pattern). By the invariance of object specifications, the object used for the update must satisfy the specification of the field to be updated. Any extra conditions that the new object may fulfil are not recorded and cannot be used later. More useful would be a "behavioural" update where result and transition specifications of the overriding method are subspecifications of the original method. This seems to be impossible as there is no notion of subspecification for store specifications.

Recursive Specifications. Recursive specifications are necessary when a field of an object or a result of one of the object's methods are supposed to satisfy the same specification as the object itself. They are needed to specify any recursive datatype. For example, if $A_{\texttt{Manager}}$ should include a list of accounts, we would need a recursive specification $\mu X.\,[\mathsf{head} : A_{\texttt{Account}}, \mathsf{tail} : X]$.

Below we discuss in more detail how recursive specifications can be dealt with in the logic.

6.2 Recursive Specifications

Syntax and Proof Rules. We introduce recursive specifications $\mu(X)A$. To prevent meaningless specifications such as $\mu(X)X$ we only allow recursion through object specifications, thereby enforcing "formal contractiveness".

$$\begin{aligned} \underline{A} &::= \top \mid \mathit{Bool} \mid [\mathsf{f}_i{:}\, {A_i}^{i=1\ldots n},\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}{T_j}^{j=1\ldots m}] \mid \mu(X)\underline{A} \\ A, B &::= \underline{A} \mid X \end{aligned}$$

where X ranges over an infinite set *TyVar* of specification variables. X is bound in $\mu(X)A$, and as usual we identify specifications up to the names of bound variables.

In addition to specification contexts Γ we introduce contexts Δ that contain specification variables with an upper bound, $X <: A$, where A is either another variable or $\top$. In the rules of the logic we replace $\Gamma \vdash \ldots$ by $\Gamma; \Delta \vdash \ldots$, and the definitions of well-formed specifications and well-formed specification contexts are extended, similar to the case of recursive types [1].

Subspecifications for recursive specifications are obtained by the "usual" recursive subtyping rule [3],

$$\frac{\Gamma; \Delta, Y <: \top, X <: Y \vdash A <: B}{\Gamma; \Delta \vdash \mu X.A <: \mu Y.B}$$

As will be seen from the semantics below, in our model a recursive specification and its unfolding are not just isomorphic but equal, i.e., $[\![\mu X.A]\!] = [\![A[(\mu X.A)/X]]\!]$. Hence we can add $\Gamma; \Delta \vdash A[(\mu X.A)/X] <: \mu X.A$ and $\Gamma; \Delta \vdash \mu X.A <: A[(\mu X.A)/X]$ and do not need to introduce *fold* and *unfold* terms.

Semantics of Recursive Specifications. We extend the interpretation of specifications to the new cases, where η maps type variables to admissible subsets of $\mathsf{Val} \times \mathsf{St}$:

$$\begin{aligned} [\![\Gamma; \Delta \vdash \top]\!]\rho\eta &= \mathsf{Val} \times \mathsf{St} \\ [\![\Gamma; \Delta \vdash X]\!]\rho\eta &= \eta(X) \\ [\![\Gamma; \Delta \vdash \mu(X)A]\!]\rho\eta &= \mathsf{gfp}(\lambda\chi.[\![\Gamma; \Delta, X <: \top \vdash A]\!]\rho\eta[X = \chi]) \end{aligned}$$

We write $\eta \vDash \Delta$ if, for all $X <: Y$ in Δ, $\eta(X) \subseteq \eta(Y)$. The set of admissible subsets of $\mathsf{Val} \times \mathsf{St}$ is closed under arbitrary intersections, hence forms a complete

lattice when ordered by set inclusion, as do environments η with the pointwise ordering $\leq$. Using well-known facts about lattices and monotonic maps one observes that the semantics preserves meets:

$$\eta_0 \geq \eta_1 \geq \ldots \Rightarrow [\![\Gamma; \Delta \vdash A]\!]\rho(\textstyle\bigwedge_i \eta_i) = \bigcap_i [\![\Gamma; \Delta \vdash A]\!]\rho\eta_i$$

In particular, the greatest fixed point in the interpretation above is guaranteed to exist, as $\mathsf{gfp}(f) = \bigcap_i f^i(\top)$ for monotonic and meet preserving f.

Existence of Store Predicates. Next, we adapt our notion of store specification to recursive specifications. A store specification is now taken to be a record $\Sigma \in \mathsf{Rec}_{\mathsf{Loc}}(\mathit{Spec})$ such that $\Sigma.l = \mu(X)[\mathsf{f}_i\colon {A_i}^{i=1\ldots n}, \mathsf{m}_j\colon \varsigma(y_j)B_j{::}{T_j}^{j=1\ldots m}]$ is a closed (recursive) object specification, for each $l \in \mathsf{dom}(\Sigma)$. Because of the (fold) and (unfold) rules, the requirement that only object specifications with a μ-binder in head position occur in Σ is no real restriction. The definition of the functional Φ of Section 4 remains virtually the same apart from an unfolding of the recursive specification in the cases for field and method result specification:

$(\sigma, \phi) \in \Phi(R, S)_\Sigma :\Leftrightarrow$
(1) ...
(2) $\forall l \in \mathsf{dom}(\Sigma)$ where $\Sigma.l = \mu(X)[\mathsf{f}_i\colon {A_i}^{i=1\ldots n}, \mathsf{m}_j\colon \varsigma(y_j)B_j{::}{T_j}^{j=1\ldots m}]$:
(F) $\sigma.l.\mathsf{f}_i \in ||A_i[\Sigma.l/X]||_\Sigma$
...
(M3) $v \in ||B_j[\Sigma.l/X, l/y_j]||_{\Sigma''}$
...

The proof of Lemma 2 can be easily adapted to show that this functional also has a unique fixpoint.

Syntactic Approximations. In Section 5, Lemma 3 was proved by induction on the structure of A. This inductive proof cannot be extended directly to prove a corresponding result for recursive specifications: The recursive unfolding in cases **(F)** and **(M3)** of the definition of $\sigma \in [\![\Sigma]\!]$ would force a similar unfolding of A in the inductive step. We consider finite approximations as in [3], where we get rid of recursion by unfolding a finite number of times and replacing all remaining occurrences of recursion by $\top$.

Definition 5 (Approximations). *For each A and $k \in \mathbb{N}$, we define $A|^k$ as*

- $A|^0 = \top$
- $\mu(X)A|^{k+1} = A[\mu(X)A/X]|^{k+1}$
- $[\mathsf{f}_i\colon {A_i}^{i=1\ldots n}, \mathsf{m}_j\colon \varsigma(y_j)B_j{::}{T_j}^{j=1\ldots m}]|^{k+1} = [(\mathsf{f}_i : A_i|^k)^{i=1\ldots n}, \mathsf{m}_j : \varsigma(y_j)B_j|^k :: {T_j}^{j=1\ldots m}]$
- $\top|^{k+1} = \top$
- $X|^{k+1} = X$
- $\mathit{Bool}|^{k+1} = \mathit{Bool}$

Lemma 5. *For all $\Gamma; \Delta \vdash A$ and $\eta \vDash \Delta$, $[\![\Gamma; \Delta \vdash A]\!]\rho\eta = \bigcap_{k\in\mathbb{N}} [\![\Gamma; \Delta \vdash A|^k]\!]\rho\eta$.*

Lemma 6. *For all $\sigma \in [\![\Sigma]\!]$, $l \in \mathsf{dom}(\Sigma)$ and (possibly recursive) A s.t. $\vdash \Sigma.l <: A$, $(l, \sigma) \in [\![A]\!]$.*

Proof. Similar to the proof of Lemma 3 one shows $(l, \sigma) \in [\![A|^k]\!]$ for all k. Then by Lemma 5, $(l, \sigma) \in \bigcap_{k\in\mathbb{N}} [\![A|^k]\!] = [\![A]\!]$.

7 Conclusion

Based on a denotational semantics, we have given a soundness proof for Abadi and Leino's program logic of an object-based language. Compared to the original proof, which was carried out wrt. an operational semantics, our techniques allowed us to distinguish the notions of derivability and validity. Further, we used the denotational framework to extend the logic to recursive object specifications. In comparison to a similar logic presented in [9] our notion of subspecification is structural rather than nominal.

Although our proof is very much different from the original one, the nature of the logic forces us to work with store specifications too. Information for locations referenced from the environment Γ will be needed for derivations. Since the Γ cannot reflect the dynamic aspect of the store (which is growing) one uses store specifications Σ. They do not show up in the Abadi-Leino logic as they are automatically preserved by programs. By contrast to [2], we can view store specifications as predicates on stores which need to be defined by mixed-variant recursion due to the form of the object introduction rule. Unfortunately, such recursively defined predicates do not directly admit an interpretation of subsumption (nor weakening).

Conditions **(M1)** – **(M3)** in the semantics of store specifications ensure that methods in the store preserve not only the current store specification but also arbitrary extensions $\Sigma' \succcurlyeq \Sigma$. Clearly, not every predicate on stores is preserved. As we lack a semantic characterisation of those specifications that are syntactically definable (as Σ), specification syntax appears in the definition of $\sigma \in [\![\Sigma]\!]$ (Def. 3). More annoyingly, field update requires subspecifications to be invariant in the field components. We do not know how to express this property of object specifications semantically (on the level of predicates) and need to use the inductively defined subspecification relation instead.

The proof of Lemma 2, establishing the existence of store predicates, provides an explanation why transition relations of the Abadi-Leino logic express properties of the flat part of stores only and allows for a quick check whether extensions are feasible. We have enumerated several extensions in Section 6.1. Based on this list and the results presented we intend to design a variation of the Abadi-Leino logic that is more expressive. We hope that this will also shed some light on modular reasoning for class-based languages.

Acknowledgement. We wish to thank Thomas Streicher for discussions and comments.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, New York, 1996.
2. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In N. Dershowitz, editor, *Verification: Theory and Practice*, pages 11–41. Springer, 2004.
3. R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

4. K. R. Apt. Ten years of Hoare's logic: A survey — part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.
5. F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *FOSSACS'99*, volume 1578 of *LNCS*, pages 135–149, 1999.
6. U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In C. Hankin, editor, *ESOP'98*, volume 1381 of *LNCS*, pages 105–121, 1998.
7. C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.
8. M. Hofmann and F. Tang. Generation of verification conditions for Abadi and Leino's logic of objects. Presented at 9th International Workshop on Foundations of Object-Oriented Languages, 2002.
9. K. R. M. Leino. Recursive object types in a logic of object-oriented programs. In C. Hankin, editor, *ESOP'98*, volume 1381 of *LNCS*, pages 170–184, 1998.
10. P. B. Levy. Possible world semantics for general storage in call-by-value. In J. Bradfield, editor, *CSL'02*, volume 2471 of *LNCS*. Springer, 2002.
11. L. C. Paulson. *Logic and Computation : Interactive proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. 1987.
12. A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.
13. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *ESOP'99*, volume 1576 of *LNCS*, pages 162–176, 1999.
14. U. S. Reddy. Objects and classes in algol-like languages. *Information and Computation*, 172(1):63–97, 2002.
15. U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1–3):129–160, 2004.
16. B. Reus. Class-based versus object-based: A denotational comparison. In H. Kirchner and C. Ringeissen, editors, *Proceedings of AMAST'02*, volume 2422 of *LNCS*, pages 473–488, 2002.
17. B. Reus. Modular semantics and logics of classes. In M. Baatz and J. A. Makowsky, editors, *CSL'03*, volume 2803 of *LNCS*, pages 456–469. Springer Verlag, 2003.
18. B. Reus and J. Schwinghammer. Denotational semantics for Abadi and Leino's logic of objects. Technical Report 2004:03, Informatics, University of Sussex, 2004.
19. B. Reus and T. Streicher. Semantics and logic of object calculi. *Theoretical Computer Science*, 316:191–213, 2004.
20. B. Reus, M. Wirsing, and R. Hennicker. A Hoare-Calculus for Verifying Java Realizations of OCL-Constrained Design Models. In H. Hussmann, editor, *FASE 2001*, volume 2029 of *LNCS*, pages 300–317, Berlin, 2001. Springer.
21. D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

A Design for a Security-Typed Language with Certificate-Based Declassification

Stephen Tse and Steve Zdancewic

University of Pennsylvania

Abstract. This paper presents a calculus that supports information-flow security policies and certificate-based declassification. The decentralized label model and its downgrading mechanisms are concisely expressed in the polymorphic lambda calculus with subtyping (System $\mathtt{F}_{\preceq}$). We prove a conditioned version of the noninterference theorem such that authorization for declassification is justified by digital certificates from public-key infrastructures.

1 Introduction

Information-flow policies constrain the propagation of confidential data and provide an end-to-end guarantee of security. *Security-typed languages* have become a promising approach for specifying and enforcing such policies with static type systems [15]. However, designing a *safe* and *secure* information-flow type system is still a challenging problem: programmers want to express fine-grained security policies with advanced types, but reasoning about security guarantees in such complex systems is non-trivial.

This paper presents a security-typed language with well-studied constructs from the polymorphic lambda calculus with subtyping (System $\mathtt{F}_{\preceq}$) [6]. Language features such as labels and effects are isolated in a monadic style. This design makes typing and evaluation rules easy to understand and the proofs of *type-safety* and *noninterference* modular.

Another challenge of designing a security-typed language is to provide downgrading mechanisms that intentionally break the security guarantees, if such actions can be justified externally. Downgrading mechanisms, such as *delegating* to another principal or *declassifying* secret data, are important in practical programming [21]. The decentralized label model by Myers and Liskov [10] addresses this problem and introduces the notions of principals and reader sets to statically track the authority for downgrading.

One of our design decisions is to treat labels, principals and downgrading privileges uniformly as types so that the decentralized label model can be integrated easily into our language. For example, subtyping naturally models principal delegation, while intersection and union types give rise to principal groups and label refinements. A security language with these encodings allows expressive, decentralized policies, yet the semantics remains easy to understand.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 279–294, 2005.

Our previous work [18] connects the static security type system with run-time security mechanisms such as public-key infrastructures. The language there uses *singleton types* such that a principal can be represented as a public key, and the authority of a principal granting a privilege is represented as a digital certificate.

We improve on our previous work by using monads and subtyping, allowing us to prove a conditioned version of noninterference even in the presence of declassification (which was neither stated nor proved before). In particular, we formalize downgrading mechanisms such as delegation and declassification as subtyping, and certificate verification as *extending* the subtyping relation. More importantly, the conditioned noninterference now captures the intuition that certificates externally justify the information leaks due to declassification.

The main contributions of our paper are:

1. the design of a safe and secure information-flow type system with bounded quantification and effects in a monadic style;
2. the integration of the decentralized label model with type constructors and the use of subtyping to model delegation, declassification, and endorsement;
3. a conditioned version of the noninterference theorem that justifies certificate-based declassification.

The work here subsumes our previous work [18], adding existential types, run-time labels and privileges, and a conditioned noninterference theorem for the full language. We have also built a prototype interpreter called *Apollo*[1].

For brevity, this paper shows only the interesting cases of rules and proof. Our technical report [17] contains all rules and proofs of type-safety and noninterference for the full language. The proof of type-safety is also mechanically formalized and checked with Twelf (a logical framework).

The rest of the paper is organized as follows. Section 2 starts with a core calculus with labels to study noninterference and extends it with effects. Section 3 introduces the decentralized label model and shows how the core calculus supports the notion of principals, confidentiality and integrity. Furthermore, downgrading mechanisms are studied as subtyping, and certificate-based declassification is justified using constructs from public-key infrastructures. The paper then discusses related work in Section 4 and concludes in Section 5.

2 Core Label Calculus

Let us start by introducing a core calculus with monadic labels and effects for analyzing program dependency. This section proves two important security theorems, *type-safety* and *noninterference*, for our core calculus.

2.1 Monadic Labels

The first part of our label calculus is based on the dependency core calculus (DCC) [1] and the polymorphic lambda calculus with subtyping (System $\mathtt{F}_{\preceq}$) [6].

[1] The interested reader is invited to visit `http://www.cis.upenn.edu/~stse/apollo`.

The motivation behind DCC is to use *monadic labels* as a unifying framework to study many important program analyses such as binding time, information flow, slicing, and function call tracking. DCC uses a lattice of monads and a special typing rule for their associated bind operations to describe the dependency of computations in a program.

Unlike DCC, which is based on the *call-by-name* simply-typed lambda calculus, our core calculus is based on the *call-by-value* $\mathtt{F}_{\preceq}$. Our work should apply also to call-by-name languages; we pick call-by-value semantics simply because of their familiarity. The features of $\mathtt{F}_{\preceq}$ will become essential in later sections: bounded quantification ($\forall\alpha \preceq \mathtt{t}.\mathtt{t}$ and $\exists\alpha \preceq \mathtt{t}.\mathtt{t}$) are used to connect static security policies and run-time public-key infrastructures (Sect. 3.3), and subtyping is used to model principal delegations and policy refinements (Sect. 3.1).

The following grammar defines the syntax for our basic types and terms:

$$\mathtt{t} ::= \mathtt{<>} \mid \mathtt{<t,t>} \mid \mathtt{t}+\mathtt{t} \mid \mathtt{t}\rightarrow\mathtt{t} \mid \top_{\mathtt{k}} \mid \bot_{\mathtt{k}} \mid \alpha \mid \forall\alpha \preceq \mathtt{t}.\mathtt{t} \mid \exists\alpha \preceq \mathtt{t}.\mathtt{t}$$

$$\begin{aligned}\mathtt{m} ::= {} & \mathtt{<>} \mid \mathtt{<m,m>} \mid \mathtt{prj}_1\ \mathtt{m} \mid \mathtt{prj}_2\ \mathtt{m} \mid \mathtt{inj}_1\ \mathtt{m} \mid \mathtt{inj}_2\ \mathtt{m} \mid \mathtt{case\ m\ m\ m} \mid \mathtt{x} \mid \lambda\mathtt{x}{:}\mathtt{t}.\mathtt{m} \mid \mathtt{m\ m} \\ & \mid \Lambda\alpha \preceq \mathtt{t}.\mathtt{m} \mid \mathtt{m\ [t]} \mid \mathtt{pack\ (t,m)\ as\ t} \mid \mathtt{open\ }(\alpha,\mathtt{x}) = \mathtt{m\ in\ m}\end{aligned}$$

The types consists of unit, products, sums, functions, top, bottom, variables, universal and existential quantification, while the terms consists of unit, products, projections, injections, cases, variables, functions, applications, type abstractions and instantiations, and package packings and openings. We also encode Booleans `bool` using unit and sums.

The types top $\top_{\mathtt{k}}$ and bottom $\bot_{\mathtt{k}}$ are annotated by a kind $\mathtt{k}$: types $\mathcal{T}$, labels $\mathcal{L}$, principals $\mathcal{P}$, and privileges $\mathcal{J}$. Principals and privileges will be explained in Sect. 3 with the decentralized label model. One of our design choices is to identify these syntactic classes (types $\mathtt{t}$, labels ℓ, principals $\mathtt{p}$, and privileges $\mathtt{j}$):

$$\mathtt{t} \equiv \ell \equiv \mathtt{p} \equiv \mathtt{j} \qquad\qquad \mathtt{k} ::= \mathcal{T} \mid \mathcal{L} \mid \mathcal{P} \mid \mathcal{J}$$

This design allows the reuse of type machinery, such as polymorphism and subtyping, uniformly for these concepts. We will see this benefit again for intersection and union types in Sect 3.1, and singleton types in Sect. 3.3.

We use the semantics of Kernel $\mathtt{F}_{\preceq}$ [6]. The evaluation judgment is denoted by $\mathtt{m} \longrightarrow \mathtt{m}$, the typing judgments by $\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}$, and the subtyping judgment by $\Delta \vdash \mathtt{t} \preceq \mathtt{t}$, where Δ is a type context and Γ is a term context. We follow Pottier's notation [13] for specifying the subtyping polarities Ⓢ of type constructors: $\oplus$ for covariant, $\ominus$ for contravariant, and $\odot$ for invariant:

$$\begin{aligned}&\Delta ::= \cdot \mid \Delta, \alpha \preceq \mathtt{t} \qquad \Gamma ::= \cdot \mid \Gamma, \mathtt{x}{:}\mathtt{t} \\ &Ⓢ ::= \mathtt{<}\oplus,\oplus\mathtt{>} \mid \oplus + \oplus \mid \ominus\rightarrow\oplus \mid \forall\alpha \preceq \odot.\oplus \mid \exists\alpha \preceq \odot.\oplus\end{aligned}$$

We omit rules for the standard $\mathtt{F}_{\preceq}$ constructs above and focus on the new types and terms for labels: monadic types $\mathtt{t}\{\ell\}$ (indexed by labels ℓ), and their corresponding units $\mathtt{m}\{\ell\}$ and `bind` operator.

$$\mathtt{t} ::= \ldots \mid \mathtt{t}\{\ell\} \qquad \mathtt{m} ::= \ldots \mid \mathtt{m}\{\ell\} \mid \mathtt{bind\ x = m\ in\ m} \qquad Ⓢ ::= \ldots \mid \oplus\{\oplus\}$$

Syntactically, these label constructs have the highest precedence, so that $\mathtt{m}_1\ \mathtt{m}_2\{\ell\}$ means $\mathtt{m}_1\ (\mathtt{m}_2\{\ell\})$. We write high and low labels as $\mathtt{H} = \top_{\mathcal{L}}$ and $\mathtt{L} = \bot_{\mathcal{L}}$.

The subtyping relation of labels $\Delta \vdash \ell \preceq \ell$ forms a lattice and hence our language has a lattice of monads $\mathtt{t}\{\ell\}$ and $\mathtt{m}\{\ell\}$. Since labels and types are in the same syntactic class, we use a kinding judgment $\Delta \vdash \mathtt{t} :: \mathtt{k}$ to rule out ill-formed types such as $\mathtt{bool} \to \mathtt{H}$ or $\mathtt{bool}\{\mathtt{bool}\}$. We omit the straight-forward kind system here; our technical report [17] contains the full details.

Now, let us see how the type system prevents low-level computation from depending on high-level computation:

$$\frac{\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t} \quad \Delta \vdash \ell :: \mathcal{L}}{\Delta;\ \Gamma \vdash \mathtt{m}\{\ell\} : \mathtt{t}\{\ell\}} \qquad \frac{\Delta;\ \Gamma \vdash \mathtt{m}_1 : \mathtt{t}_1\{\ell\} \quad \Delta;\ \Gamma, \mathtt{x}{:}\mathtt{t}_1 \vdash \mathtt{m}_2 : \mathtt{t}_2 \quad \Delta \vdash \ell \ll \mathtt{t}_2}{\Delta;\ \Gamma \vdash \mathtt{bind}\ \mathtt{x} = \mathtt{m}_1\ \mathtt{in}\ \mathtt{m}_2 : \mathtt{t}_2}$$

The label monad $\mathtt{m}\{\ell\}$ marks the computation $\mathtt{m}$ with the label ℓ, restricting how it interacts with the rest of the program. The term $\mathtt{bind}\ \mathtt{x} = \mathtt{m}_1\ \mathtt{in}\ \mathtt{m}_2$ exposes the computation $\mathtt{m}_1$ protected inside the label type $\mathtt{t}\{\ell\}$ to the scope of $\mathtt{m}_2$.

Note that these typings are standard for monadic types, except that the return type of `bind` here has type $\mathtt{t}_2$, rather than the expected type $\mathtt{t}_2\{\ell\}$. Instead, by connecting the subtyping of labels ($\Delta \vdash \ell_1 \preceq \ell_2$) with the subtyping of types ($\Delta \vdash \mathtt{t}_1 \preceq \mathtt{t}_2$), the following label protection judgment $\Delta \vdash \ell \ll \mathtt{t}$ ensures that the result of `bind` still protects the data:

$$\Delta \vdash \ell \ll \langle\rangle \qquad \frac{\Delta \vdash \ell_2 \preceq \ell_1}{\Delta \vdash \ell_2 \ll \mathtt{t}\{\ell_1\}} \qquad \frac{\Delta \vdash \ell_2 \ll \mathtt{t}}{\Delta \vdash \ell_2 \ll \mathtt{t}\{\ell_1\}}$$

The unit type protects all labels as there is only one term of such type. Sum types, as information can be leaked by their tags, do not protect any label. The full set of rules also includes cases for products, functions, and universal types.

Example 1. The term $\lambda \mathtt{x}{:}\mathtt{bool}\{\mathtt{H}\}.\ \mathtt{bind}\ \mathtt{y} = \mathtt{x}\ \mathtt{in}\ \mathtt{if}\ \mathtt{y}\ \mathtt{then}\ 0\ \mathtt{else}\ 1$ is *not* well-typed, because $\Delta \vdash \mathtt{H} \not\ll \mathtt{int}$. An integer leaks information just like a Boolean or a sum. In contrast, $\lambda \mathtt{x}{:}\mathtt{bool}\{\mathtt{H}\}.\ \mathtt{bind}\ \mathtt{y} = \mathtt{x}\ \mathtt{in}\ \mathtt{if}\ \mathtt{y}\ \mathtt{then}\ 0\{\mathtt{H}\}\ \mathtt{else}\ 1\{\mathtt{H}\}$ *is* well-typed, because $\Delta \vdash \mathtt{H} \ll \mathtt{int}\{\mathtt{H}\}$. □

Operationally, the label monad $\mathtt{m}\{\ell\}$ evaluates the term inside until it is a value $\mathtt{v}\{\ell\}$, while `bind` evaluates $\mathtt{m}_1$ to a value $\mathtt{v}\{\ell\}$ and substitutes $\mathtt{v}$ for $\mathtt{x}$ in $\mathtt{m}_2$. We specify the dynamic semantics by the following syntactic classes of values $\mathtt{v}$ and evaluation contexts $\mathtt{E}$ [20], and by small-step computation rules. We use $\mathtt{m}\{\mathtt{v}/\mathtt{x}\}$ to denote the capture-free substitution of $\mathtt{v}$ for $\mathtt{x}$ in $\mathtt{m}$.

$$\begin{array}{ll} \mathtt{v} ::= \ldots \mid \mathtt{v}\{\ell\} & \\ \mathtt{E} ::= \ldots \mid \mathtt{E}\{\ell\} \mid \mathtt{bind}\ \mathtt{x} = \mathtt{E}\ \mathtt{in}\ \mathtt{m} & \mathtt{bind}\ \mathtt{x} = \mathtt{v}\{\ell\}\ \mathtt{in}\ \mathtt{m} \longrightarrow \mathtt{m}\{\mathtt{v}/\mathtt{x}\} \end{array}$$

DCC also has fixpoints and pointed types. In the technical report, we add such features to the full language and prove noninterference using a bisimulation-like technique. For the lack of space, however, such development is left out here.

2.2 Security Theorems

Before we go on to enrich the language with features such as effects and the decentralized label model, let us state and prove two important theorems that guarantee the security of programs written in our language. These theorems still

hold for our full languages (modulo some condition to account for declassification, to be explained in Sect. 3.3); however, we prove them here for the core calculus first to demonstrate the proof techniques. By presenting and proving for the full language incrementally, we hope to substantiate our claim that monadic types make the design and proofs more modular.

The first theorem is the type-safety of the language, which we have proved using the progress and preservation theorems. Type-safety states that a closed, well-typed program will not get stuck or generate any error. A closed program means that both the type and term contexts are empty, that is, $\Delta = \Gamma = \cdot$.

Theorem 2 (Progress and Preservation). If $\cdot;\ \cdot \vdash \mathtt{m}_1 : \mathtt{t}$, then either $\mathtt{m}_1 = \mathtt{v}$ or $\mathtt{m}_1 \longrightarrow \mathtt{m}_2$. And, if $\Delta;\ \Gamma \vdash \mathtt{m}_1 : \mathtt{t}$ and $\mathtt{m}_1 \longrightarrow \mathtt{m}_2$, then $\Delta;\ \Gamma \vdash \mathtt{m}_2 : \mathtt{t}$.

Proof. By induction on the typing derivation [6, 1]. □

The second theorem is the noninterference property of the language [15], which states that if a program is well-typed, a low-level observer cannot distinguish between different high-level computations. The theorem requires a model of *observers* ζ for specifying what information leaks are possible. Our model here is that, given an equivalence relation over values of the same type, a well-typed observer cannot distinguish *equivalent values*, which are parameterized by the security label of the observer.

For example, we should have these equivalences for Booleans:

$$\mathtt{true} \sim_\zeta \mathtt{true} : \mathtt{bool} \qquad \mathtt{true} \not\sim_\zeta \mathtt{false} : \mathtt{bool} \qquad \mathtt{true}\{\mathtt{H}\} \sim_\mathtt{L} \mathtt{false}\{\mathtt{H}\} : \mathtt{bool}\{\mathtt{H}\}$$

The first two say that no observer ζ cannot distinguish `true` from `true`, but an observer can tell the difference between `true` and `false`. More interestingly, the third says that if values are protected inside the high monad, then different values become indistinguishable to the low-level observer `L`.

Based on the intuition above, we generalize the equivalence relation in the following ways: (1) extend the relation to be higher-order, to account for functions; (2) parameterize the relation with arbitrary labels; (3) cover all types and values in the relation; and, (4) lift the relation from values to terms by evaluation.

Formally, this logical equivalence relation is defined by the following rules. We denote the equivalence relation for closed values at closed type `t` by $\mathtt{v} \sim_\zeta \mathtt{v} : \mathtt{t}$, and that for closed terms by $\mathtt{m} \approx_\zeta \mathtt{m} : \mathtt{t}$:

$$\frac{\mathtt{m}_1 \longrightarrow^* \mathtt{v}_1 \quad \mathtt{m}_2 \longrightarrow^* \mathtt{v}_2 \quad \mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \mathtt{t}}{\mathtt{m}_1 \approx_\zeta \mathtt{m}_2 : \mathtt{t}} \qquad \frac{\forall(\mathtt{v}_3 \sim_\zeta \mathtt{v}_4 : \mathtt{t}_1).\ \mathtt{v}_1\ \mathtt{v}_3 \approx_\zeta \mathtt{v}_2\ \mathtt{v}_4 : \mathtt{t}_2}{\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \mathtt{t}_1 \to \mathtt{t}_2}$$

$$\frac{\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \mathtt{t}}{\mathtt{v}_1\{\ell\} \sim_\zeta \mathtt{v}_2\{\ell\} : \mathtt{t}\{\ell\}} \qquad \frac{\forall(\mathtt{t}_2 \preceq \mathtt{t}_1).\ \mathtt{v}_1\ [\mathtt{t}_2] \approx_\zeta \mathtt{v}_2\ [\mathtt{t}_2] : \mathtt{t}\{\mathtt{t}_2/\alpha\}}{\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \forall\alpha \preceq \mathtt{t}_1.\ \mathtt{t}}$$

$$\frac{\ell \not\preceq \zeta}{\mathtt{v}_1\{\ell\} \sim_\zeta \mathtt{v}_2\{\ell\} : \mathtt{t}\{\ell\}} \qquad \frac{\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \mathtt{t}\{\mathtt{t}_1/\alpha\}}{\mathtt{pack}\ (\mathtt{t}_1, \mathtt{v}_1) \sim_\zeta \mathtt{pack}\ (\mathtt{t}_1, \mathtt{v}_2) : \exists\alpha \preceq \mathtt{t}_2.\mathtt{t}}$$

For reference in proofs later, we name these rules (from top to bottom, left to right) R-Term, R-Lab1, R-Lab2, R-Fun, R-All, and R-Some (with type annotations inside the `pack` terms elided). We slightly abuse the notation by using

$\forall$ both for the object-level quantification types $\forall\alpha \preceq \mathtt{t.t}$ and for the meta-level quantification in logical relations. Note that we do not deal with parametricity of polymorphic functions [19] nor the behavioral equivalence of existential packages [12]. That is, our model assumes that an observer can differentiate different representations of polymorphic functions or different implementations of existential packages. This assumption simplifies the equivalence relations, and is the key difference between noninterference and parametricity.

The last step is to model an arbitrary observer as an open term that contains free type variables and term variables, and model observations as type substitutions δ and term substitutions γ, which are defined as:

$$\delta ::= \cdot \mid \delta, \alpha \mapsto \mathtt{t} \qquad\qquad \gamma ::= \cdot \mid \gamma, \mathtt{x} \mapsto \mathtt{v}$$

A judgment $\delta \models \Delta$ says that a type substitution models a type context: for all $\alpha \in \mathtt{dom}(\delta) = \mathtt{dom}(\Delta)$, if $\delta(\alpha) = \mathtt{t}_1$ and $\alpha \preceq \mathtt{t}_2 \in \Delta$, then $\mathtt{t}_1$ is closed, has the same kind as $\mathtt{t}_2$, and $\Delta \vdash \mathtt{t}_1 \preceq \mathtt{t}_2$. Another judgment $\gamma_1 \sim_\zeta \gamma_2 : \delta(\Gamma)$ says that two term substitutions are equivalent under a term context of closed types: for all $\mathtt{x} \in \mathtt{dom}(\gamma_1) = \mathtt{dom}(\gamma_2) = \mathtt{dom}(\delta(\Gamma))$, if $\gamma_1(\mathtt{x}) = \mathtt{v}_1$, $\gamma_2(\mathtt{x}) = \mathtt{v}_2$ and $\mathtt{x}{:}\mathtt{t} \in \delta(\Gamma)$, then $\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \mathtt{t}$.

With the logical relations and the substitutions above, we can formally state the main theorem of the core label calculus: related substitutions preserve the logical equivalence. In other words, an arbitrary observer cannot distinguish values higher in the lattice.

Theorem 3 (Noninterference for Terms). If $\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}$ and $\delta \models \Delta$ and $\gamma_1 \sim_\zeta \gamma_2 : \delta(\Gamma)$, then $\delta\gamma_1(\mathtt{m}) \approx_\zeta \delta\gamma_2(\mathtt{m}) : \delta(\mathtt{t})$.

Proof. By induction on the typing derivation. Case `bind`: We are given $\Delta;\ \Gamma \vdash \mathtt{bind\ x = m_1\ in\ m_2} : \mathtt{t}_2$. By inversion, we have $\Delta;\ \Gamma \vdash \mathtt{m}_1 : \mathtt{t}_1\{\ell\}$ (*1) and $\Delta;\ \Gamma, \mathtt{x}:\mathtt{t}_1 \vdash \mathtt{m}_2 : \mathtt{t}_2$ (*2) and $\Delta \vdash \ell \ll \mathtt{t}_2$ (*3). By induction hypothesis with (*1), we have

$$\delta\gamma_1(\mathtt{m}_1) \approx_\zeta \delta\gamma_2(\mathtt{m}_1) : \delta(\mathtt{t}_1\{\ell\})$$

By inversion of R-Term, $\delta\gamma_1(\mathtt{m}_1) \longrightarrow^* \mathtt{v}_1$ (*4) and $\delta\gamma_2(\mathtt{m}_1) \longrightarrow^* \mathtt{v}_2$ (*5) and $\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \delta(\mathtt{t}_1\{\ell\})$. Subcase $\delta(\ell) \preceq \zeta$: By the inversion of R-Lab1, $\mathtt{v}_1 = \mathtt{v}_3\{\delta(\ell)\}$ and $\mathtt{v}_2 = \mathtt{v}_4\{\delta(\ell)\}$ and $\mathtt{v}_3 \sim_\zeta \mathtt{v}_4 : \delta(\mathtt{t}_1)$ (*6). We then extend the term substitutions as

$$\gamma_1' = \gamma_1, \mathtt{x} \mapsto \mathtt{v}_3 \qquad \gamma_2' = \gamma_2, \mathtt{x} \mapsto \mathtt{v}_4$$

such that, by (*6), $\gamma_1' \sim_\zeta \gamma_2' : \delta(\Gamma, \mathtt{x} : \mathtt{t}_1)$ (*7). By induction hypothesis with (*2,*7),

$$\delta\gamma_1'(\mathtt{m}_2) \approx_\zeta \delta\gamma_2'(\mathtt{m}_2) : \delta(\mathtt{t}_2)$$

which means that $\delta\gamma_1(\mathtt{m}_2)\{\mathtt{v}_3/\mathtt{x}\} \approx_\zeta \delta\gamma_2(\mathtt{m}_2)\{\mathtt{v}_4/\mathtt{x}\} : \delta(\mathtt{t}_2)$ (*8). By (*4,*5),

$$\begin{array}{rl} & \delta\gamma_1(\mathtt{bind\ x = m_1\ in\ m_2}) \\ = & \mathtt{bind\ x} = \delta\gamma_1(\mathtt{m}_1)\ \mathtt{in}\ \delta\gamma_1(\mathtt{m}_2) \\ \longrightarrow^* & \mathtt{bind\ x} = \mathtt{v}_3\{\delta(\ell)\}\ \mathtt{in}\ \delta\gamma_1(\mathtt{m}_2) \\ \longrightarrow & \delta\gamma_1(\mathtt{m}_2)\{\mathtt{v}_3/\mathtt{x}\} \end{array} \qquad \begin{array}{rl} & \delta\gamma_2(\mathtt{bind\ x = m_1\ in\ m_2}) \\ = & \mathtt{bind\ x} = \delta\gamma_2(\mathtt{m}_1)\ \mathtt{in}\ \delta\gamma_2(\mathtt{m}_2) \\ \longrightarrow^* & \mathtt{bind\ x} = \mathtt{v}_4\{\delta(\ell)\}\ \mathtt{in}\ \delta\gamma_2(\mathtt{m}_2) \\ \longrightarrow & \delta\gamma_2(\mathtt{m}_2)\{\mathtt{v}_4/\mathtt{x}\} \end{array}$$

Therefore, by R-Term and (*8), we conclude that $\delta\gamma_1(\mathtt{bind\ x = m_1\ in\ m_2}) \approx_\zeta \delta\gamma_2(\mathtt{bind\ x = m_1\ in\ m_2}) : \delta(\mathtt{t}_2)$. Subcase $\delta(\ell) \not\preceq \zeta$: by Lemma 4 with (*3). □

Lemma 4 (Noninterference for Protected Terms). If $\Delta \vdash \ell \ll \mathtt{t}$, $\delta \models \Delta$ and $\delta(\ell) \not\preceq \zeta$, then $\mathtt{m}_1 \approx_\zeta \mathtt{m}_2 : \mathtt{t}$. □

2.3 Monadic Effects

We now turn to study information flows in the presence of computational effects. Practical programs interact with external systems and produce effects; observers can then learn high-security values from those effects. To prevent information leaks through such channel, we need to refine the type system with effect types.

We again use the monadic style of effect types [9, 5]. The benefit of monads is that the new feature can be incrementally added to the language we have shown so far. That is, all the typing and evaluation rules in Section 2.1 remain unchanged. Traditional approaches, in contrast, require tracking of effects in all typing rules, spreading the interaction of labels and effects everywhere. Monads also help in structuring proofs in a modular way, which will be explained for Theorem 7.

For lazy languages like Haskell, we can simply add the IO monad. For eager languages like the one here, we need to introduce a new syntactic class $\mathtt{e}$ for *effectful expressions* to distinguish from *pure terms* $\mathtt{m}$ introduced in Sect. 2.1:

$$\begin{array}{ll} \mathtt{t} ::= \ldots \mid \mathtt{t!}\epsilon & \textcircled{\$} ::= \ldots \mid \oplus\,!\ominus \\ \mathtt{e} ::= \mathtt{return\ m} \mid \mathtt{run\ x = m\ in\ e} & \mathtt{m} ::= \ldots \mid \mathtt{e!}\epsilon \end{array}$$

Every top-level program is now an expression, instead of a term. We model effects as outputs at a given label ℓ, which are visible to an observer of level ζ if $\ell \preceq \zeta$. An observer cannot tell the difference between effects of different labels, but can count the number of visible effects happening in the program. This treatment gives us a uniform way of modeling language features with effects and could be extended to effects that carry additional values. Experience with effectful languages suggests that this technique can be extended to memory references with reads and writes [5].

Expressions are $\mathtt{return\ m}$ and $\mathtt{run\ x = m\ in\ e}$, which explicitly specify the order of execution. The term $\mathtt{e!}\epsilon$ delays the effects ϵ in $\mathtt{e}$ and thus can be considered pure, but has the monadic effect type $\mathtt{t!}\epsilon$ (indexed by effect labels ϵ). Here ϵ is a lower bound on the labels of observable effects happening in $\mathtt{e}$, similar to the concept of *program counter label* in the literature [15].

The typing judgment for expressions is $\Delta;\ \Gamma \vdash \mathtt{e} : \mathtt{t!}\epsilon$, which says that under the type context Δ and term context Γ, the expression $\mathtt{e}$ has the monadic effect type $\mathtt{t!}\epsilon$. The following are the typing rules for the new constructs:

$$\frac{\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}}{\Delta;\ \Gamma \vdash \mathtt{return\ m} : \mathtt{t!H}} \qquad \frac{\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}_1\mathtt{!}\epsilon \quad \Delta;\ \Gamma, \mathtt{x}{:}\mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2\mathtt{!}\epsilon}{\Delta;\ \Gamma \vdash \mathtt{run\ x = m\ in\ e} : \mathtt{t}_2\mathtt{!}\epsilon}$$

$$\frac{\Delta;\ \Gamma \vdash \mathtt{e} : \mathtt{t!}\epsilon \quad \Delta \vdash \epsilon :: \mathcal{L}}{\Delta;\ \Gamma \vdash \mathtt{e!}\epsilon : \mathtt{t!}\epsilon} \qquad \frac{\Delta \vdash \ell \ll \mathtt{t} \quad \Delta \vdash \ell \preceq \epsilon}{\Delta \vdash \ell \ll \mathtt{t!}\epsilon}$$

The expression $\mathtt{return\ m}$ has no effect and hence its type is given the empty effect $\mathtt{H}$. We interpret the effect at $\mathtt{H}$ to be visible to no-one, while the effect at $\mathtt{L}$ to be visible to everyone. The expression $\mathtt{run\ x = m\ in\ e}$ executes the

encapsulated effect of m, and then continue with e. Both m and e have the same effect type ϵ; otherwise, the subsumption rule of subtyping can be used.

The bottom left rule simply connects the typing judgments of terms and expressions. The bottom right rule, on the other hand, is an additional label protection judgment (defined in Sect. 2) for effect types. The rule says that the underlying type must protect the label and the computation must generate effects higher than the label. In other words, once the program has bound high-security data, it may not produce low observable effects.

Example 5. The expression $\texttt{run z} = \texttt{(bind y} = \texttt{x in if y then c!H else c!L) in z}$, where $\texttt{c} \equiv \texttt{return <>}$ and $\texttt{x:bool\{H\}}$, is insecure. This is a typical example of *implicit information flow through program counter* in the literature [15], where a program leaks information about a high-security Boolean through side effects.

The evaluation judgment for expressions is $\texttt{e} \xrightarrow{\epsilon} \texttt{e}$, where ϵ is the side effect during such step. We use u to denote the values for expressions:

$$\texttt{u} ::= \texttt{return v} \qquad \texttt{v} ::= \ldots \mid \texttt{e!}\epsilon$$
$$\texttt{E} ::= \ldots \mid \texttt{return E} \mid \texttt{run x} = \texttt{E in e} \mid \texttt{run x} = \texttt{(return E)!}\epsilon \texttt{ in e}$$
$$\texttt{run x} = \texttt{(return v)!}\epsilon \texttt{ in e} \xrightarrow{\epsilon} \texttt{e\{v/x\}}$$

Since the congruence rules for expressions have no computational effects, we can still use evaluation contexts E to describe the evaluation order of expressions. The term $\texttt{e!}\epsilon$ is a value because it is a closure that delays computation.

Example 6. The following expression of type bool!L evaluates as:

$$\begin{array}{ll} & \texttt{run x} = \texttt{(run y} = \texttt{(return prj}_2\ \texttt{<true, false>)!L in return y)!H in return x} \\ \longrightarrow & \texttt{run x} = \texttt{(run y} = \texttt{(return false)!L in return y)!H in return x} \\ \xrightarrow{\texttt{L}} & \texttt{run x} = \texttt{(return y)\{false/y\}!H in return x} \\ = & \texttt{run x} = \texttt{(return false)!H in return x} \\ \xrightarrow{\texttt{H}} & \texttt{(return x)\{false/x\}!H} \\ = & \texttt{return false} \end{array} \qquad \square$$

To model that an observer can now distinguish programs due to computational effects, we need the following new equivalence judgments for effectful expressions and values: $\texttt{e} \approx_\zeta \texttt{e} : \texttt{t!}\epsilon$ and $\texttt{u} \sim_\zeta \texttt{u} : \texttt{t!}\epsilon$. The rules for expressions make use of a new evaluation relation, $\texttt{e} \stackrel{\zeta}{\Longrightarrow}^{\texttt{n}} \texttt{u}$, which is explained below.

$$\frac{\texttt{e}_1 \approx_\zeta \texttt{e}_2 : \texttt{t!}\epsilon}{\texttt{e}_1\texttt{!}\epsilon \sim_\zeta \texttt{e}_2\texttt{!}\epsilon : \texttt{t!}\epsilon}\ \text{(R-Eff)} \qquad \frac{\texttt{v}_1 \sim_\zeta \texttt{v}_2 : \texttt{t}}{\texttt{return v}_1 \sim_\zeta \texttt{return v}_2 : \texttt{t!}\epsilon}\ \text{(R-Ret)}$$

$$\frac{\texttt{e}_1 \stackrel{\zeta}{\Longrightarrow}^{\texttt{n}} \texttt{u}_1 \quad \texttt{e}_2 \stackrel{\zeta}{\Longrightarrow}^{\texttt{n}} \texttt{u}_2 \quad \texttt{u}_1 \sim_\zeta \texttt{u}_2 : \texttt{t!}\epsilon}{\texttt{e}_1 \approx_\zeta \texttt{e}_2 : \texttt{t!}\epsilon}\ \text{(R-Exp)}$$

The rules on the top simply connect the term equivalence and the expression equivalence. Expressions are equivalent, the bottom rule says, if they produce the same number of effects visible to the observer and halt at equivalent values.

To formalize such equivalence, we first classify evaluation steps into those that are *visible* and those that are *invisible* to the observer. Then, a visible evaluation step can be prefixed and suffixed with any number of invisible evaluation steps.

$$\xrightarrow{\preceq\zeta} \equiv \bigcup_{\epsilon\preceq\zeta} \xrightarrow{\epsilon} \qquad \xrightarrow{\not\preceq\zeta} \equiv \bigcup_{\epsilon\not\preceq\zeta} \xrightarrow{\epsilon} \qquad \stackrel{\zeta}{\Longrightarrow} \equiv \xrightarrow{\not\preceq\zeta}{}^{*} \circ \xrightarrow{\preceq\zeta} \circ \xrightarrow{\not\preceq\zeta}{}^{*}$$

The evaluation judgment we want is therefore the $\mathtt{n}$-step closure $\stackrel{\zeta}{\Longrightarrow}{}^{\mathtt{n}}$ of $\stackrel{\zeta}{\Longrightarrow}$. Note that $\xrightarrow{\not\preceq\zeta}{}^{*} \circ \xrightarrow{\preceq\zeta}$ is the composition of the two relations, while $\xrightarrow{\not\preceq\zeta}{}^{*}$ is the reflexive and transitive closure of $\xrightarrow{\not\preceq\zeta}$.

Having refined our observer model as above, we proceed to proving noninterference for our core calculus with expressions. The main idea is to track the number of visible effects produced during the evaluation.

Note that the following proof is complete yet short in length. Since the proof is by induction on the typing derivation, monadic types allow an incremental proof, because the original proof for Theorem 3 remains valid and requires only a simple extension for $\mathtt{e!}\epsilon$. Here we can focus merely on the new typing rules for `return e` and `run x = m in e`.

Theorem 7 (Noninterference for Expressions). If $\Delta;\ \Gamma \vdash \mathtt{e} : \mathtt{t!}\epsilon$ and $\delta \models \Delta$ and $\gamma_1 \sim_\zeta \gamma_2 : \delta(\Gamma)$, then $\delta\gamma_1(\mathtt{e}) \approx_\zeta \delta\gamma_2(\mathtt{e}) : \delta(\mathtt{t})!\delta(\epsilon)$.

Proof. By mutual induction with Theorem 3 (extended with $\mathtt{e!}\epsilon$) on the typing derivation. Case `return`: We are given $\Delta;\ \Gamma \vdash \mathtt{return\ m} : \mathtt{t!H}$. By inversion, we have $\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}$. By Theorem 3, we have $\delta\gamma_1(\mathtt{m}) \approx_\zeta \delta\gamma_2(\mathtt{m}) : \delta(\mathtt{t})$. By inversion of R-Term, $\delta\gamma_1(\mathtt{m}) \longrightarrow^* \mathtt{v}_1$ and $\delta\gamma_2(\mathtt{m}) \longrightarrow^* \mathtt{v}_2$ and $\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \delta(\mathtt{t})$. Therefore, by R-Ret, we conclude that $\delta\gamma_1(\mathtt{return\ m}) \approx_\zeta \delta\gamma_2(\mathtt{return\ m}) : \delta(\mathtt{t})!\delta(\epsilon)$.

Case `run`: We are given $\Delta;\ \Gamma \vdash \mathtt{run\ x = m\ in\ e} : \mathtt{t}_2\mathtt{!}\epsilon$. By inversion, we have $\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}_1\mathtt{!}\epsilon$ (*1) and $\Delta;\ \Gamma, \mathtt{x}{:}\mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2\mathtt{!}\epsilon$ (*2). By Theorem 3 with (*1), we have

$$\delta\gamma_1(\mathtt{m}) \approx_\zeta \delta\gamma_2(\mathtt{m}) : \delta(\mathtt{t}_1)!\delta(\epsilon)$$

By inversion of R-Term, $\delta\gamma_1(\mathtt{m}) \longrightarrow^* \mathtt{v}_1$ (*3) and $\delta\gamma_2(\mathtt{m}) \longrightarrow^* \mathtt{v}_2$ (*4) and $\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \delta(\mathtt{t}_1)!\delta(\epsilon)$. By inversion of R-Eff, $\mathtt{v}_1 = \mathtt{e}_1!\delta(\epsilon)$ and $\mathtt{v}_2 = \mathtt{e}_2!\epsilon$ and $\mathtt{e}_1 \approx_\zeta \mathtt{e}_2 : \delta(\mathtt{t}_1)!\delta(\epsilon)$. By inversion of R-Exp, $\mathtt{e}_1 \stackrel{\zeta}{\Longrightarrow}{}^{\mathtt{n}} \mathtt{u}_1$ (*5) and $\mathtt{e}_2 \stackrel{\zeta}{\Longrightarrow}{}^{\mathtt{n}} \mathtt{u}_2$ (*6) and $\mathtt{u}_1 \sim_\zeta \mathtt{u}_2 : \delta(\mathtt{t}_1) : \delta(\epsilon)$. By inversion of R-Ret, $\mathtt{u}_1 = \mathtt{return\ v}_3!\delta(\epsilon)$ and $\mathtt{u}_2 = \mathtt{return\ v}_4!\delta(\epsilon)$. We then extend the term substitutions as

$$\gamma_1' = \gamma_1, \mathtt{x} \mapsto \mathtt{v}_3 \qquad \gamma_2' = \gamma_2, \mathtt{x} \mapsto \mathtt{v}_4$$

such that $\gamma_1' \sim_\zeta \gamma_2' : \delta(\Gamma, \mathtt{x}{:}\mathtt{t}_1)$ (*7). By induction hypothesis with (*2,*7),

$$\delta\gamma_1'(\mathtt{e}) \approx_\zeta \delta\gamma_2'(\mathtt{e}) : \delta(\mathtt{t}_2)!\delta(\epsilon)$$

which means that $\delta\gamma_1(\mathtt{e})\{\mathtt{v}_3/\mathtt{x}\} \approx_\zeta \delta\gamma_2(\mathtt{e})\{\mathtt{v}_4/\mathtt{x}\} : \delta(\mathtt{t}_2)!\delta(\epsilon)$ (*8). By (*3,*4,*5,*6),

$$\begin{array}{ll}
 & \delta\gamma_1(\mathtt{run\ x = m\ in\ e}) \\
= & \mathtt{run\ x} = \delta\gamma_1(\mathtt{m})\ \mathtt{in}\ \delta\gamma_1(\mathtt{e}) \\
\longrightarrow^* & \mathtt{run\ x} = \mathtt{e}_1!\delta(\epsilon)\ \mathtt{in}\ \delta\gamma_1(\mathtt{e}) \\
\stackrel{\zeta}{\Longrightarrow}{}^{\mathtt{n}} & \mathtt{run\ x} = (\mathtt{return\ v}_3)!\delta(\epsilon)\ \mathtt{in}\ \delta\gamma_1(\mathtt{e}) \\
\xrightarrow{\epsilon} & \delta\gamma_1(\mathtt{e})\{\mathtt{v}_3/\mathtt{x}\}
\end{array}
\qquad
\begin{array}{ll}
 & \delta\gamma_2(\mathtt{run\ x = m\ in\ e}) \\
= & \mathtt{run\ x} = \delta\gamma_2(\mathtt{m})\ \mathtt{in}\ \delta\gamma_2(\mathtt{e}) \\
\longrightarrow^* & \mathtt{run\ x} = \mathtt{e}_2!\delta(\epsilon)\ \mathtt{in}\ \delta\gamma_2(\mathtt{e}) \\
\stackrel{\zeta}{\Longrightarrow}{}^{\mathtt{n}} & \mathtt{run\ x} = (\mathtt{return\ v}_4)!\delta(\epsilon)\ \mathtt{in}\ \delta\gamma_2(\mathtt{e}) \\
\xrightarrow{\epsilon} & \delta\gamma_2(\mathtt{e})\{\mathtt{v}_4/\mathtt{x}\}
\end{array}$$

That means

$$\delta\gamma_1(\texttt{run x} = \texttt{m in e}) \stackrel{\zeta}{\Longrightarrow}^{\texttt{n+i}} \delta\gamma_1(\texttt{e})\{\texttt{v}_3/\texttt{x}\}$$
$$\delta\gamma_2(\texttt{run x} = \texttt{m in e}) \stackrel{\zeta}{\Longrightarrow}^{\texttt{n+i}} \delta\gamma_2(\texttt{e})\{\texttt{v}_4/\texttt{x}\}$$

where $\texttt{i} = 1$ if $\epsilon \preceq \zeta$, or $\texttt{i} = 0$ otherwise. By R-Exp and (*8), we conclude that $\delta\gamma_1(\texttt{run x} = \texttt{m in e}) \approx_\zeta \delta\gamma_2(\texttt{run x} = \texttt{m in e}) : \delta(\texttt{t}_2)!\delta(\epsilon)$. □

3 Decentralized Label Calculus

Having established the security property of our core calculus, we now investigate how to make the policy sublanguage more expressive. The key challenge is to extend the policy language in a modular way, reusing the type machinery from the core as much as possible.

This section shows how the decentralized label model by Myers and Liskov [10] can be integrated into our core label calculus. Decentralized labels allow different *principals* to individually specify fine-grained security policies such as *confidentiality* and *integrity*. Combined with *singleton types*, this extended calculus draws a connection between compile-time dependency analyses and the run-time infrastructure. The benefit is twofold: (1) security policies can now be specified in term of information not known until execution, such as run-time user identities or file access permissions; (2) certificates can be used to regulate declassification and to justify a conditioned version of the noninterference theorem.

3.1 Confidentiality and Integrity

Confidentiality policies specify which principals allow which other principals to *read* some data, while integrity policies specify which principals *trust* some data [7]. These policy constructors, or *label constructors*, provide a finer-grained control of security specification than the *label constants* introduced in Sect. 2.1.

To model these policies, we treat principals $\texttt{p}$ as abstract types and treat principal delegation $\texttt{p}_1 \preceq \texttt{p}_2$ as subtyping. That is, $\texttt{p}_1$ is a subtype of $\texttt{p}_2$ whenever $\texttt{p}_1$ *delegates to* $\texttt{p}_2$ (or, $\texttt{p}_2$ is *acting for* $\texttt{p}_1$). We also introduce two new label constructors, $\texttt{R}$ (read) and $\texttt{T}$ (trust), for confidentiality and integrity:

$$\ell ::= \ldots \mid \texttt{R p p} \mid \texttt{T p} \mid \ell \wedge \ell \mid \ell \vee \ell \qquad \textcircled{S} ::= \ldots \mid \texttt{R} \oplus \oplus \mid \texttt{T} \ominus \mid \oplus \wedge \oplus \mid \oplus \vee \oplus$$

A label $\texttt{R}\ \texttt{p}_1\ \texttt{p}_2$ specifies the policy that a data is owned by $\texttt{p}_1$ and that $\texttt{p}_1$ allows $\texttt{p}_2$ to read the data, while a label $\texttt{T p}$ specifies that the data is trusted by $\texttt{p}$.

Moreover, we add intersection $\ell \wedge \ell$ and union types $\ell \vee \ell$ [3] to precisely model policy sets. Since labels ℓ and principals $\texttt{p}$ are in the same syntactic class, these two constructors can also model principal groups as $\texttt{p} \wedge \texttt{p}$ and $\texttt{p} \vee \texttt{p}$.

Intersection and union types in this paper are used only for labels, principals, and privileges, but not for ordinary types; hence, our language does not have introduction or elimination terms for intersections and unions. This decision helps keeping the static and the dynamic semantics of our language simple.

We need both intersection and union types because the two label constructors have different subtyping polarities: $\texttt{R}$ is covariant, while $\texttt{T}$ is contravariant. Having both intersections and unions gives a natural interpretation of principal sets:

$$\begin{aligned}\mathtt{R}\ [\mathtt{p}_1,\mathtt{p}_2,\ldots,\mathtt{p}_n]\ \mathtt{p} &= \mathtt{R}\ (\mathtt{p}_1 \wedge \mathtt{p}_2 \wedge \ldots \wedge \mathtt{p}_n)\ \mathtt{p}\\ \mathtt{R}\ \mathtt{p}\ [\mathtt{p}_1,\mathtt{p}_2,\ldots,\mathtt{p}_n] &= \mathtt{R}\ \mathtt{p}\ (\mathtt{p}_1 \wedge \mathtt{p}_2 \wedge \ldots \wedge \mathtt{p}_n)\\ \mathtt{T}\ [\mathtt{p}_1,\mathtt{p}_2,\ldots,\mathtt{p}_n] &= \mathtt{T}\ (\mathtt{p}_1 \vee \mathtt{p}_2 \vee \ldots \vee \mathtt{p}_n)\end{aligned}$$

Example 8. The data $\mathtt{true}\{\mathtt{R}\ \mathtt{p}_1\ [\mathtt{p}_2,\mathtt{p}_3]\}\{\mathtt{T}\ [\mathtt{p}_1,\mathtt{p}_2]\}$ has two security policies. The first one is a confidentiality policy saying that the data is owned by $\mathtt{p}_1$, and that $\mathtt{p}_1$ allows $\mathtt{p}_2$ and $\mathtt{p}_3$ to read the data. The second one is an integrity policy saying that both $\mathtt{p}_1$ and $\mathtt{p}_2$ trust the data. □

A decentralized label looks like $\{\mathtt{p}_1 : \mathtt{p}_2,\mathtt{p}_3;\ \mathtt{p}_2 : \mathtt{p}_3\ !\ \mathtt{p}_1,\mathtt{p}_2\}$ traditionally [10, 18], compared to our notation $\{\mathtt{R}\ \mathtt{p}_1\ [\mathtt{p}_2,\mathtt{p}_3]\}\{\mathtt{R}\ \mathtt{p}_2\ \mathtt{p}_3\}\{\mathtt{T}\ [\mathtt{p}_1\ \mathtt{p}_2]\}$ here. Ours is slightly more verbose but its semantics can be specified more easily in terms of subtyping. In addition, new policy constructors can be added in a uniform way by simply specifying their subtyping polarities.

3.2 Downgrading as Subtyping

The rest of the section discusses various downgrading mechanisms that intentionally leak information [21]. These mechanisms include:

1. *declassifying* some data to a lower label,
2. a principal *delegating* to other principals,
3. a principal *declassifying* some data to other principals for reading, and
4. a principal *endorsing* the integrity of some data.

The decentralized label model is essential in the last three mechanisms because each concerns a particular *principal*. In Sect. 3.3 we will see how a public key, which represents the concerned principal, can be used to verify a digital certificate, which represents the authority for downgrading.

The innovation here is to model downgrading as subtyping. The motivation is that downgrading can be made *implicit* through the subsumption rule of subtyping, if the concerned principal *explicitly* introduces the authority into the context. This contrasts with the usual approach [10] that uses coercion constructs like $\mathtt{declassify}_{\mathtt{p}}\ \mathtt{m}$ and $\mathtt{endorse}_{\mathtt{p}}\ \mathtt{m}$ for declassification and endorsement. Both approaches ensure that the authority of the concerned principal is granted before declassification. Our implicit approach, however, allows a simple formulation of certificate-based declassification (to be shown in Sect. 3.3).

Foremost, we extend the type context Δ to maintain authority, which is a set of authorizations of the form $\mathtt{p} \lhd \mathtt{j}$ (a principal $\mathtt{p}$ *granting* some privilege $\mathtt{j}$):

$$\begin{array}{ll}\Delta ::= \ldots \mid \Delta, \mathtt{p} \lhd \mathtt{j} & \\ \mathtt{j} ::= \ldots \mid \mathtt{del}\ \mathtt{p} \mid \mathtt{dcls}\ \mathtt{p} \mid \mathtt{endr} & \text{Ⓢ} ::= \ldots \mid \mathtt{del} \oplus \mid \mathtt{dcls} \oplus \mid \oplus\ \mathtt{\%} \ominus \lhd \oplus \\ \mathtt{t} ::= \ldots \mid \mathtt{t}\ \mathtt{\%}\ \mathtt{p} \lhd \mathtt{j} & \mathtt{m} ::= \ldots \mid \mathtt{grant}\ \mathtt{p} \lhd \mathtt{j}\ \mathtt{in}\ \mathtt{m} \mid \mathtt{pass}\ \mathtt{x} = \mathtt{m}\ \mathtt{in}\ \mathtt{m}\end{array}$$

The three predefined privileges are delegation ($\mathtt{del}\ \mathtt{p}$), declassification ($\mathtt{dcls}\ \mathtt{p}$), and endorsement ($\mathtt{endr}$), corresponding to downgrading for principal subtyping, confidentiality and integrity, respectively. Now, the downgrading mechanisms can be concisely expressed using these additional subtyping rules:

$$\frac{\Delta \vdash \mathtt{p}_1 \lhd \mathtt{del}\ \mathtt{p}_2}{\Delta \vdash \mathtt{p}_1 \preceq \mathtt{p}_2} \qquad \frac{\Delta \vdash \mathtt{p}_1 \lhd \mathtt{dcls}\ \mathtt{p}}{\Delta \vdash \mathtt{R}\ \mathtt{p}_1\ \mathtt{p}_2 \preceq \mathtt{R}\ \mathtt{p}_1\ [\mathtt{p}_2, \mathtt{p}]} \qquad \frac{\Delta \vdash \mathtt{p} \lhd \mathtt{endr}}{\Delta \vdash \mathtt{T}\ \mathtt{p}_1 \preceq \mathtt{T}\ [\mathtt{p}_1, \mathtt{p}]}$$

An *authority type* $\mathtt{t}\ \%\ \mathtt{p} \lhd \mathtt{j}$ (a type $\mathtt{t}$ annotated with the authority of a principal $\mathtt{p}$ granting some privilege $\mathtt{j}$) tracks the *effects* of declassification on the lattice so that later theorems can be stated in terms of the authority:

$$\frac{\Delta, \mathtt{p} \lhd \mathtt{j};\ \Gamma \vdash \mathtt{m} : \mathtt{t} \quad \Delta \vdash \mathtt{p} :: \mathcal{P} \quad \Delta \vdash \mathtt{j} :: \mathcal{J}}{\Delta;\ \Gamma \vdash \mathtt{grant}\ \mathtt{p} \lhd \mathtt{j}\ \mathtt{in}\ \mathtt{m} : \mathtt{t}\ \%\ \mathtt{p} \lhd \mathtt{j}} \qquad \frac{\begin{array}{c}\Delta;\ \Gamma \vdash \mathtt{m}_1 : \mathtt{t}_1\ \%\ \mathtt{p} \lhd \mathtt{j} \\ \Delta, \mathtt{p} \lhd \mathtt{j};\ \Gamma, \mathtt{x}{:}\mathtt{t}_1 \vdash \mathtt{m}_2 : \mathtt{t}_2\end{array}}{\Delta;\ \Gamma \vdash \mathtt{pass}\ \mathtt{x} = \mathtt{m}_1\ \mathtt{in}\ \mathtt{m}_2 : \mathtt{t}_2\ \%\ \mathtt{p} \lhd \mathtt{j}}$$

$$\mathtt{v} ::= \ldots \mid \mathtt{grant}\ \mathtt{p} \lhd \mathtt{j}\ \mathtt{in}\ \mathtt{v} \qquad \mathtt{E} ::= \ldots \mid \mathtt{grant}\ \mathtt{p} \lhd \mathtt{j}\ \mathtt{in}\ \mathtt{E} \mid \mathtt{pass}\ \mathtt{x} = \mathtt{E}\ \mathtt{in}\ \mathtt{e}$$

$$\mathtt{pass}\ \mathtt{x} = (\mathtt{grant}\ \mathtt{p} \lhd \mathtt{j}\ \mathtt{in}\ \mathtt{v})\ \mathtt{in}\ \mathtt{m} \longrightarrow \mathtt{grant}\ \mathtt{p} \lhd \mathtt{j}\ \mathtt{in}\ \mathtt{m}\{\mathtt{v}/\mathtt{x}\}$$

These rules are very close to the typing and evaluation rules for standard monadic types, except that the type context Δ is now extended with $\mathtt{p} \lhd \mathtt{j}$. The value $\mathtt{v}$ in the term $\mathtt{grant}\ \mathtt{p} \lhd \mathtt{j}\ \mathtt{in}\ \mathtt{v}$ may capture the constraint $\mathtt{p} \lhd \mathtt{j}$, and hence, to ensure type preservation, $\mathtt{grant}\ \mathtt{p} \lhd \mathtt{j}\ \mathtt{in}\ \mathtt{v}$ is regarded together as a value.

As a pleasant bonus of monadic analysis, checking the *robustness condition* of downgrading reduces to adding one condition in the label protection rule $\Delta \vdash \ell \ll \mathtt{t}$ in Sect. 2.1. In particular, robust declassification says that the program context of a declassification operation should be trusted by the owner of the data [21]. The following rule generalizes the robustness condition to any downgrading mechanism. The intuition is that, for robust downgrading, when $\mathtt{p}_2$ authorizes some privilege $\mathtt{j}$, the program context should have trust $(\mathtt{T}\ \mathtt{p}_1)$ higher than $\mathtt{p}_2$'s trust $(\mathtt{T}\ \mathtt{p}_2)$. That is $\Delta \vdash \mathtt{T}\ \mathtt{p}_2 \preceq \mathtt{T}\ \mathtt{p}_1$, or equivalently, $\Delta \vdash \mathtt{p}_1 \preceq \mathtt{p}_2$.

$$\frac{\Delta \vdash \mathtt{T}\ \mathtt{p}_1 \ll \mathtt{t} \quad \Delta \vdash \mathtt{p}_1 \preceq \mathtt{p}_2}{\Delta \vdash \mathtt{T}\ \mathtt{p}_1 \ll (\mathtt{t}\ \%\ \mathtt{p}_2 \lhd \mathtt{j})}$$

It is known that noninterference does not hold in the presence of downgrading [15]. Yet, it is intuitive that if the program does not use any downgrading, the program should still be secure. In fact, a slightly stronger statement holds: if no one transitively downgrades to the observer, the program is still secure.

The following modified theorem of noninterference formally captures such intuition. We write $\Delta = \Delta_\alpha, \Delta_\lhd$ to separate the bindings and the authority, and we write $\mathtt{t} \Rightarrow \mathtt{t}_0\ \%\ \Delta$ to collect all required authority in the value positions of the type. For example, $\mathtt{p}_1, \mathtt{j}, \mathtt{p}_1 \lhd \mathtt{j}, \mathtt{p}_2 \vdash \mathtt{m} : \mathtt{bool} \to (\mathtt{bool}\ \%\ \mathtt{p}_1 \lhd \mathtt{j})$ has $\Delta_\alpha = \mathtt{p}_1, \mathtt{j}, \mathtt{p}_2$ and $\Delta_\lhd = \mathtt{p}_1 \lhd \mathtt{j}$ and $\mathtt{bool} \to (\mathtt{bool}\ \%\ \mathtt{p}_1 \lhd \mathtt{j}) \Rightarrow (\mathtt{bool} \to \mathtt{bool})\ \%\ \mathtt{p}_1 \lhd \mathtt{j}$. These straight-forward rules are defined in our technical report [17].

Theorem 9 (Conditioned Noninterference). Suppose $\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}$, where $\Delta = \Delta_\alpha, \Delta_\lhd$ and $\mathtt{t} \Rightarrow \mathtt{t}_0\ \%\ \Delta_0$, and $\delta \models \Delta_\alpha$ and $\gamma_1 \sim_\zeta \gamma_2 : \delta(\Gamma)$. If $\Delta, \Delta_0 \not\vdash \mathtt{p} \preceq \zeta$ for all $\mathtt{p} \in \mathtt{dom}(\Delta_\alpha)$ such that $\Delta \not\vdash \mathtt{p} \preceq \zeta$, then $\delta\gamma_1(\mathtt{m}) \approx_\zeta \delta\gamma_2(\mathtt{m}) : \delta(\mathtt{t})$.

Proof. By induction on the typing derivation. Case $\lambda \mathtt{x}{:}\mathtt{t}_1.\mathtt{m}$: We are given $\Delta;\ \Gamma \vdash \lambda \mathtt{x} : \mathtt{t}_1.\mathtt{m} : \mathtt{t}_1 \to \mathtt{t}_2$. By inversion, we have $\Delta;\ \Gamma, \mathtt{x} : \mathtt{t}_1 \vdash \mathtt{m} : \mathtt{t}_2$ (*1). Since downgrading in the input propagates to the output in a function, we have $\mathtt{t}_2 \Rightarrow$

$\mathtt{t}_3 \,\%\, \Delta_0$ (*2) for the same Δ_0 as in $\mathtt{t}_1 \to \mathtt{t}_2 \Rightarrow \mathtt{t}_0 \,\%\, \Delta_0$. Assume $\mathtt{v}_3 \sim_\zeta \mathtt{v}_4 : \delta(\mathtt{t}_1)$. We then extend the term substitutions as

$$\gamma_1' = \gamma_1, \mathtt{x} \mapsto \mathtt{v}_3 \qquad\qquad \gamma_2' = \gamma_2, \mathtt{x} \mapsto \mathtt{v}_4$$

such that $\gamma_1' \sim_\zeta \gamma_2' : \delta(\Gamma, \mathtt{x}{:}\mathtt{t}_1)$ (*3). By induction hypothesis with (*1,*2,*3),

$$\delta\gamma_1'(\mathtt{m}) \approx_\zeta \delta\gamma_2'(\mathtt{m}) : \delta(\mathtt{t}_2)$$

which, by R-Term, $\delta\gamma_1(\mathtt{m})\{\mathtt{v}_3/\mathtt{x}\} \approx_\zeta \delta\gamma_2(\mathtt{m})\{\mathtt{v}_4/\mathtt{x}\} : \delta(\mathtt{t}_2)$. By R-Term again,

$$\delta\gamma_1(\lambda \mathtt{x}{:}\mathtt{t}.\mathtt{m})\ \mathtt{v}_3 \approx_\zeta \delta\gamma_2(\lambda \mathtt{x}{:}\mathtt{t}.\mathtt{m})\ \mathtt{v}_4 : \delta(\mathtt{t}_2)$$

Hence, by R-Fun, $\delta\gamma_1(\lambda \mathtt{x}{:}\mathtt{t}.\mathtt{m}) \approx_\zeta \delta\gamma_2(\lambda \mathtt{x}{:}\mathtt{t}.\mathtt{m}) : \delta(\mathtt{t}_1 \to \mathtt{t}_2)$. □

The proof is surprisingly similar to that for standard noninterference, showing that this conditioned version is only a slight generalization. In the next subsection, however, we will show how combining this theorem with ideas from public-key infrastructures justifies certificate-based declassification.

3.3 Public Keys and Certificates

Public-key infrastructures provide public keys and digital certificates for distributed access control. Our motivation here is to connect the type system with the security infrastructure such that a certificate of authority, when verified with a principal's public key, can justify the information leaks due to downgrading. Certificates are also important for auditing purpose.

In our previous work [18], we presented the language λ_{RP} for specifying security policies with run-time principals. The type system uses *singleton types* to represent run-time principals and an abstract type to represent certificates. Effectively, λ_{RP} models public keys and certificate verifications of public-key infrastructures in a sound type system.

Allowing such run-time principals gives programmers more flexibility in specifying security policies. Together with universal and existential quantification, programs can determine the run-time user identity of the system (`getuid`) and write functions polymorphic in principals (`getenv`). Here the type $\top_{\mathcal{P}}$ represents the top principal, and $'\alpha$ is a singleton type to be explained below.

$$\begin{array}{lcl} \mathtt{getuid} & : & ()\to\exists\alpha \preceq \top_{\mathcal{P}}.\ '\alpha \\ \mathtt{getenv} & : & \forall\alpha \preceq \top_{\mathcal{P}}.\ '\alpha \to \mathtt{string}\{\mathtt{R}\ \alpha\ \alpha\} \end{array}$$

Our language readily generalizes the idea of *run-time types* to run-time labels and run-time privileges as well. For example, access permissions from the file system (`fstat`) can be used as run-time labels to constrain the information flow of data read from a file.

Let us recap our previous work on run-time principals [18]:

$$\mathtt{t} ::= \ldots \mid \mathtt{'p} \mid \mathtt{'j} \mid \mathtt{cert} \qquad \mathtt{m} ::= \ldots \mid \mathtt{'p} \mid \mathtt{'j} \mid \mathtt{'p} \lhd \mathtt{'j} \mid \mathtt{if\ (m} \Rightarrow \mathtt{m} \lhd \mathtt{m)\ m\ m}$$

The term `'p` is the run-time representation of principal `p` and has the singleton type `'p`, carrying the most precise information about the term in the type system. Similarly, `'j` is the run-time representation of privilege `j`.

The term $\texttt{'p} \triangleleft \texttt{'j}$ represents the authority of principal $\texttt{p}$ granting privilege $\texttt{j}$. Such term has the abstract type $\texttt{cert}$ that does not reveal any information at all at the type level. The reason is that we do not trust the validity of the certificate unless verified with the term $\texttt{if (}\mathtt{m}_1 \Rightarrow \mathtt{m}_2 \triangleleft \mathtt{m}_3\texttt{)}\ \mathtt{m}_4\ \mathtt{m}_5$. More formally:

$$\Delta;\ \Gamma \vdash \texttt{'p} : \texttt{'p} \qquad \Delta;\ \Gamma \vdash \texttt{'j} : \texttt{'j} \qquad \Delta;\ \Gamma \vdash \texttt{'p} \triangleleft \texttt{'j} : \texttt{cert}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{m}_1 : \texttt{cert} \quad \Delta; \Gamma \vdash \mathtt{m}_2 : \texttt{'p} \quad \Delta; \Gamma \vdash \mathtt{m}_3 : \texttt{'j} \quad \Delta, \mathtt{p} \triangleleft \mathtt{j};\ \Gamma \vdash \mathtt{m}_4 : \mathtt{t} \quad \Delta;\ \Gamma \vdash \mathtt{m}_5 : \mathtt{t}}{\Delta;\ \Gamma \vdash \texttt{if (}\mathtt{m}_1 \Rightarrow \mathtt{m}_2 \triangleleft \mathtt{m}_3\texttt{)}\ \mathtt{m}_4\ \mathtt{m}_5 : \mathtt{t}\ \texttt{\%}\ \mathtt{p} \triangleleft \mathtt{j}}$$

$$\frac{\vdash \texttt{'}\mathtt{p}_1 \triangleleft \texttt{'}\mathtt{j}_1 \Rightarrow \texttt{'}\mathtt{p}_2 \triangleleft \texttt{'}\mathtt{j}_2}{\texttt{if ('}\mathtt{p}_1 \triangleleft \texttt{'}\mathtt{j}_1 \Rightarrow \texttt{'}\mathtt{p}_2 \triangleleft \texttt{'}\mathtt{j}_2\texttt{)}\ \mathtt{m}_1\ \mathtt{m}_2 \longrightarrow \texttt{grant}\ \mathtt{p}_2 \triangleleft \mathtt{j}_2\ \texttt{in}\ \mathtt{m}_1}$$

The judgment $\vdash \texttt{'}\mathtt{p}_1 \triangleleft \texttt{'}\mathtt{j}_1 \Rightarrow \texttt{'}\mathtt{p}_2 \triangleleft \texttt{'}\mathtt{j}_2$ defines an external verification procedure of the authority with respect to the principal and the privilege. If the procedure fails, the term $\texttt{if ('}\mathtt{p}_1 \triangleleft \texttt{'}\mathtt{j}_1 \Rightarrow \texttt{'}\mathtt{p}_2 \triangleleft \texttt{'}\mathtt{j}_2\texttt{)}\ \mathtt{m}_1\ \mathtt{m}_2$ steps to $\mathtt{m}_2$.

There exists a direct mapping from the language constructs ($\texttt{'p}$, $\texttt{'p} \triangleleft \texttt{'j}$, and $\vdash \texttt{'}\mathtt{p}_1 \triangleleft \texttt{'}\mathtt{j}_1 \Rightarrow \texttt{'}\mathtt{p}_2 \triangleleft \texttt{'}\mathtt{j}_2$) to the mechanisms of public-key infrastructures (public keys and digital certificates). In fact, public-key infrastructures are just one possible implementation that supports distributed access control [4]. Our previous work [18] carries out the design and the proof in an abstract setting and provides constructs for testing delegation and acquiring certificates.

We conclude the development of our language by presenting a modified theorem of noninterference. It states that any information leaked by a well-typed program can be justified by certificates in the environment. In fact, the theorem is simply the contrapositive of the conditioned noninterference in Sect. 3.2. Our technical report [17] contains detailed proofs of the type-safety and the following theorem for the full language.

Theorem 10 (Certified Noninterference). Suppose $\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}$, where $\Delta = \Delta_\alpha, \Delta_\triangleleft$ and $\mathtt{t} \Rightarrow \mathtt{t}_0\ \texttt{\%}\ \Delta_0$, and $\delta \models \Delta_\alpha$ and $\gamma_1 \sim_\zeta \gamma_2 : \delta(\Gamma)$. If $\delta\gamma_1(\mathtt{m}) \not\approx_\zeta \delta\gamma_2(\mathtt{m}) : \delta(\mathtt{t})$, then $\mathtt{t} = \mathtt{t}_0\ \texttt{\%}\ \Delta_0$ and $\Delta, \Delta_0 \vdash \mathtt{p} \preceq \zeta$ for some $\mathtt{p}$ that satisfies $\Delta \not\vdash \mathtt{p} \preceq \zeta$.

Proof. By Theorem 9, extended with singletons and certificates.

4 Related Work

The survey by Sabelfeld and Myers [15] on language-based information-flow security is an excellent introduction to the field. In particular, their paper cites a long line of research [10, 13, 2] that studies the interactions of security policies and language features in Java and ML. This paper instead focuses on a smaller set of interesting features with a modular design and with the goal of justifying declassification with certificates. Compared to our previous work [18], this paper concisely expresses the decentralized label model in the polymorphic lambda calculus with subtyping ($\mathtt{F}_\preceq$). Various downgrading mechanisms are understood as

subtyping such that not only type-safety but also a conditioned version of noninterference can be formulated and proved. We also extensively employ monadic constructs [9, 1, 5] to keep the design and the proofs modular. As a future work, one may check if these constructs satisfy some formal monad laws.

Chothia et al. also use public-key infrastructures to model typed cryptographic operations for distributed access control [4]. Strecker [16] formalizes an analysis of information flow for μ-Java and proves noninterference in Isabelle by shallow embedding, while Naumann [11] similarly formalizes a core subset of Java in PVS by deep embedding. Our ongoing work has the same goal of proving noninterference in a machine-checkable way.

It is known that standard noninterference does not hold in the presence of declassification [15]. Hence, it has been a challenging problem to formulate and prove *any* variant of noninterference with declassification. Various ideas such as *selective declassification* [13], *delimited release* [14], and *relaxed noninterference* [8] are proposed to allow downgrading that can be externally justified.

5 Conclusion

We have presented the design of a safe and secure information-flow type system with bounded quantification and effects in a monadic style. One of our design decisions is to treat labels, principals and privileges uniformly, as they are all abstract types necessary only for compile-time analyses. This treatment allows reuse of type machinery such as polymorphism, subtyping, and singleton types, keeping the calculus consistent yet general.

The integration of the decentralized label model with type constructors allows programmers specify expressive policies, while the use of subtyping to model delegation, declassification, and endorsement simplifies the semantics of downgrading. More importantly, these simplifications lead to a conditioned version of the noninterference theorem that justifies certificate-based downgrading.

Formalizing the full language semantics and security theorems is our long-term goal of building a rigid foundation for security-typed languages. One exciting future work is to use Twelf (a logical framework) to mechanically formalize and check the various noninterference theorems presented in this paper. We are also writing larger examples in our language interpreter to gain more experience of monadic secure programming.

Acknowledgments

The authors thank Peng Li, Nitin Khandelwal, Eijiro Sumii, and the anonymous reviewers for their comments on drafts of this paper. This research was supported in part by NSF grant CCR-0311204 *(Dynamic Security Policies)* and NSF grant CNS-0346939 *(CAREER: Language-based Distributed System Security)*.

References

1. Martin Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A Core Calculus of Dependency. In *ACM Symposium on Principles of Programming Languages*, 1999.
2. Anindya Banerjee and David A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Computer Security Foundations Workshop*, 2002.
3. Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and Union Types: Syntax and Semantics. *Information and Computation*, 119, 1995.
4. Tom Chothia, Dominic Duggan, and Jan Vitek. Type-Based Distributed Access Control. In *Computer Security Foundations Workshop*, 2003.
5. Karl Crary, Aleksey Kliger, and Frank Pfenning. A monadic analysis of info flow security with mutable state. In *Foundations of Computer Security*, 2004.
6. Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in Fsub. *Mathematical Structures in Computer Science*, 1992.
7. Peng Li, Yun Mao, and Steve Zdancewic. Information Integrity Policies. In *Proceedings of the Workshop on Formal Aspects in Security and Trust (FAST)*, 2003.
8. Peng Li and Steve Zdancewic. Downgrading Policies and Relaxed Noninterference. In *ACM Symposium on Principles of Programming Languages*, 2004.
9. Eugenio Moggi. Computational Lambda-Calculus and Monads. In *IEEE Symposium on Logic in Computer Science*, 1989.
10. Andrew C. Myers and Barbara Liskov. A Decentralized Model for Information Flow Control. In *ACM Symposium on Operating Systems Principles*, 1997.
11. David A. Naumann. Machine-checked correctness of a secure information flow analyzer. Technical Report CS-2004-10, Stevens Institute of Technology, 2004.
12. Andrew Pitts. Existential Types: Logical Relations and Operational Equivalence. In *International Colloquium on Automata, Languages and Programming*, 1998.
13. Francois Pottier and Vincent Simonet. Information flow inference for ML. In *ACM Symposium on Principles of Programming Languages*, 2002.
14. Andrei Sabelfeld and Andrew C. Myers. A Model for Delimited Release. In *International Symposium on Software Security*, 2003.
15. Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.
16. Martin Strecker. Formal Analysis of an Information Flow Type System for MicroJava. Technical report, Technische Universitat Munchen, 2003.
17. Stephen Tse and Steve Zdancewic. Certificate-based Declassification. Technical Report MS-CIS-04-16, University of Pennsylvania, 2004.
18. Stephen Tse and Steve Zdancewic. Run-time Principals in Information-flow Type Systems. In *IEEE Symposium on Security and Privacy*, 2004.
19. Philip Wadler. Theorems for Free! In *Functional Programming Languages and Computer Architecture*, 1989.
20. Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1), 1994.
21. Steve Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 1997.

Adjoining Declassification and Attack Models by Abstract Interpretation

Roberto Giacobazzi and Isabella Mastroeni

Dipartimento di Informatica,
Università di Verona,
Strada Le Grazie 15, I-37134 Verona, Italy
{roberto.giacobazzi, mastroeni}@sci.univr.it

Abstract. In this paper we prove that attack models and robust declassification in language-based security can be viewed as adjoint transformations of abstract interpretations. This is achieved by interpreting the well known Joshi and Leino's semantic approach to non-interference as a problem of making an abstraction complete relatively to a program's semantics. This observation allows us to prove that the most abstract property on confidential data which flows, here called private observation, and the most concrete harmless attacker observing public data, here called public observable, both modeled as abstractions of the program's semantics, are respectively the adjoint solutions of a completeness problem in standard abstract interpretation theory. In particular declassification corresponds to refining the given model of an attacker with the minimal amount of information in order to achieve completeness, which is non-interference, while the harmless attacker corresponds to remove this information. This proves an adjunction relation between two basic approaches to language-based security: declassification and the construction of suitable attack models, and allows us to apply relevant techniques for abstract domain transformation in language-based security.

Keywords: Abstract interpretation, language-based security, declassification, abstract non-interference, attack models, adjunction, completeness.

1 Introduction

Many security problems in language-based security are problems of confidentiality: If a user wants to keep some information confidential then he/she has to state a policy stipulating that no data visible from other users is affected by confidential data. This policy allows programs to manipulate private data, unless visible/public outputs of those programs do not improperly reveal information about the data [27]. The usual way used to show confidentiality is to prove that an attacker cannot observe any difference between the public outputs of any two executions differing only in their private inputs with the assumption that an attacker (or unauthorized user) is allowed to view only information that is

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 295–310, 2005.

not confidential. In this case the program is said to satisfy *non-interference* [18], also referred as *secrecy* [27, 30]. In standard non-interference, the attacker can fully analyze concrete computations. In this case, any conservative type/data-flow/control-flow analysis of information flows would discard all the programs which may provide any explicit or implicit concrete flows from confidential to public resources. Standard non-interference is therefore often too strict for practical use in language-based security. In order to adapt security policies to practical cases, it would be essential to know how much an attacker may learn from a program by (statically) analyzing its input/output behavior. This idea has recently lead to the definition of the notion of *abstract non-interference* [13] and robust declassification [32]. Abstract non-interference provides a method for modeling attackers as abstract interpretations [7, 8] of the input/output program behavior, in particular it is used for characterizing the most powerful attacker which is not able to disclose confidential properties, in the following called *harmless*. Declassification corresponds to downgrade the sensitivity of data in order to accommodate with (intentional) information leakage. In [13] and [32] systematic methods have been designed for deriving and analyzing respectively attack models and declassification by characterizing what information flows from confidential to public variables. It is clear that the stronger is the attacker, the more information can be released by the program. Namely, the more concrete is the model of the harmless attacker, the more abstract is the confidential information that can be kept private. This observation gives an intuitive explanation of the adjoint relation existing between the actions of weakening attackers and declassifying private information. In particular, we can note that when we derive the most concrete attack model, then we are looking for the most concrete *public observer*, while when we derive the most abstract property the flows during computation, for characterizing abstract declassification, we are looking for the most abstract *private observable*. Indeed, the most concrete public observer is the model of the most powerful attacker that can observe only public data. While, the most abstract private observable is the minimal amount of information that a program releases during computation.

In this paper, we prove that this duality corresponds precisely to an adjunction in the lattice of abstract interpretations. This is achieved by considering abstract non-interference as a generalization of both declassification for passive attackers and attack models. In this setting we prove that, under non restrictive hypotheses, abstract non-interference corresponds precisely to making abstract interpretation complete (see [17]) relatively to the denotational semantics of programs. This derives directly from an abstract interpretation-based generalization of Joshi and Leino's approach to secure information flows [19], which makes this approach equivalent to a completeness problem. Abstract interpretation plays a key role here, providing the adequate framework where program properties can be compared by considering their relative precision. In particular, we prove that declassification and attack models are adjoint notions and they correspond respectively to the minimal complete refinement, providing the most concrete *public observer* property of the program and the minimal complete simplification, providing the most abstract *private observable* property of the program.

2 Basic Notions

If S and T are sets, then $\wp(S)$ denotes the powerset of S, $S \times T$ denotes the Cartesian product of S and T, $S \smallsetminus T$ denotes the set-difference between S and T, $S \subsetneq T$ denotes strict inclusion, and for a function $f : S \to T$ and $X \subseteq T$, $f(X) \stackrel{\text{def}}{=} \{f(x) \mid x \in X\}$ and $f^{-1}(X) \stackrel{\text{def}}{=} \left\{ x \,\middle|\, f(x) \in X \right\}$. We will often denote $f(\{x\})$ as $f(x)$ and use lambda notation for functions. Function composition $\lambda x.\ f(g(x))$ is denoted $f \circ g$. $\langle P, \leq \rangle$ denotes a poset P with ordering relation $\leq$, while $\langle P, \leq, \vee, \wedge, \top, \bot \rangle$ denotes a complete lattice P, with ordering $\leq$, *lub* $\vee$, *glb* $\wedge$, greatest element (top) $\top$, and least element (bottom) $\bot$. Often, $\leq_P$ will be used to denote the underlying ordering of a poset P, and $\vee_P$, $\wedge_P$, $\top_P$ and $\bot_P$ denote the basic operations and elements if P is a complete lattice. $\mathtt{id} \stackrel{\text{def}}{=} \lambda x.\ x$ and $\mathbb{T} \stackrel{\text{def}}{=} \lambda x.\ \top$. If $S \subseteq P$ then $\downarrow S \stackrel{\text{def}}{=} \{x \in P \mid \exists y \in S.\ x \leq y\}$. $\downarrow x$ is a shorthand for $\downarrow \{x\}$. $f : C \to A$ is (completely) additive if f preserves *lub*'s of all subsets of C (emptyset included). Continuity holds when f preserved *lubs*'s of chains. Co-additivity and co-continuity are dually defined.

It is well known that abstract domains can be equivalently formulated either in terms of Galois connections or closure operators [8]. A pair of functions $\alpha : C \to A$ and $\gamma : A \to C$ on posets, denoted $\langle C, \alpha, A, \gamma \rangle$, forms an *adjunction* or a *Galois connection* (GC) if for any $x \in C$ and $y \in A$: $\alpha(x) \leq_A y \Leftrightarrow x \leq_C \gamma(y)$. α (resp. γ) is the *left-* (*right-*)*adjoint* to γ (α) and it is an additive (co-additive) function. Additive and co-additive functions f admit respectively right and left adjoint: $f^+ \stackrel{\text{def}}{=} \lambda x.\ \bigvee \left\{ y \,\middle|\, f(y) \leq x \right\}$ and $f^- \stackrel{\text{def}}{=} \lambda x.\ \bigwedge \left\{ y \,\middle|\, x \leq f(y) \right\}$ respectively. Remember that $(f^+)^- = (f^-)^+ = f$ [2]. If in addition for any $a \in A$: $\alpha(\gamma(a)) = a$, then we call $\langle C, \alpha, A, \gamma \rangle$ a Galois insertion (GI) of A in C. In GC-based abstract interpretation the concrete and abstract domains, C and A, are complete lattices [7]. An *upper (lower) closure operator* $\rho : P \to P$ on a poset P is monotone, idempotent, and extensive: $\forall x \in P.\ x \leq_P \rho(x)$ (reductive: $\forall x \in P.\ x \geq_P \rho(x)$). The set of all upper (lower) closure operators on P is denoted by $uco(P)$ ($lco(P)$). Let $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$ be a complete lattice. Closure operators are uniquely determined by the set of their fix-points $\rho(C)$. For upper closures, $X \subseteq C$ is the set of fix-points of $\rho \in uco(C)$ iff X is a *Moore-family* of C, i.e., $X = \mathcal{M}(X) \stackrel{\text{def}}{=} \{\wedge S \mid S \subseteq X\}$ — where $\wedge \varnothing = \top \in \mathcal{M}(X)$, iff X is isomorphic to an abstract domain A in a GI (C, α, A, γ), i.e., $A \cong \rho(C)$ with $\iota : \rho(C) \to A$ and $\iota^{-1} : A \to \rho(C)$ being an isomorphism, and $(C, \iota \circ \rho, A, \iota^{-1})$ is the GI, i.e., $\rho = \gamma \circ \alpha$. In this case $\rho(C)$ is a complete sub-lattice of C iff ρ is additive. Dual properties can be derived for lower closures. Therefore $uco(C)$ is isomorphic to the so called *lattice of abstract interpretations of* C [8]. If C is a complete lattice then $uco(C)$ and $lco(C)$ ordered point-wise are also complete lattices. For upper closures $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \mathbb{T}, \mathtt{id} \rangle$ where for every $\rho, \eta \in uco(C)$, $\{\rho_i\}_{i \in I} \subseteq uco(C)$ and $x \in C$: $\rho \sqsubseteq \eta$ iff $\forall y \in C.\ \rho(y) \leq \eta(y)$ iff $\eta(C) \subseteq \rho(C)$; $(\sqcap_{i \in I} \rho_i)(x) = \wedge_{i \in I} \rho_i(x)$; and $(\sqcup_{i \in I} \rho_i)(x) = x \ \Leftrightarrow\ \forall i \in I.\ \rho_i(x) = x$. Dual properties can be derived for $\langle lco(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\ x, \lambda x.\ \bot \rangle$. In the following we will find particularly convenient to identify closure operators (and therefore abstract domains) with their sets of fix-points. The *disjunctive completion* of an abstract

domain $\rho \in uco(C)$ is the most abstract domain able to represent the concrete disjunction of its objects: $\curlyvee(\rho) = \sqcup\{\eta \in uco(C) | \eta \sqsubseteq \rho \text{and } \eta \text{ is additive}\}$. ρ is disjunctive iff $\curlyvee(\rho) = \rho$ (cf. [8]). Closure operators and partitions are related concepts. If π is a partition (viz. an equivalence relation), then $[\cdot]_\pi$ is the corresponding equivalence class. A closure $\eta \in uco(\wp(S))$ induces a partition on S: $\{\ [x]_\eta \,|\, x \in S\ \}$, where $[x]_\eta \stackrel{\text{def}}{=} \{\ y \,|\, \eta(x) = \eta(y)\ \}$. The most concrete closure that induces the same partition of values as η is $\mathcal{P}(\eta) \stackrel{\text{def}}{=} \curlyvee(\{\ [x]_\eta \,|\, x \in S\ \})$. η is *partitioning* if $\eta = \mathcal{P}(\eta)$ [24]. The idea is that $\mathcal{P}(\eta)$ is the most concrete closure such that for any $y \in \mathcal{P}(\eta(x))$: $\mathcal{P}(\eta(x)) = \mathcal{P}(\eta(y))$, while in general $\eta(y) \subseteq \eta(x)$.

In abstract interpretation there are two equivalent ways to express the soundness of an abstraction [7]. Let C be a complete lattice, $f : C \to C$, (C, α, A, γ) be a Galois insertion, and $f^\sharp : A \to A$. Then (C, α, A, γ) and $f^\sharp$ provide a sound abstraction of f if $\alpha \circ f \leq f^\sharp \circ \alpha$, or equivalently (by adjunction) if $f \circ \gamma \leq \gamma \circ f^\sharp$. While these two definitions of soundness are equivalent, they are not equivalent when equality is required, i.e., when we consider completeness [8, 17, 15]. In the first case $\alpha \circ f = f^\sharp \circ \alpha$ means that no loss of precision is accumulated by approximating the input arguments of a given semantic function; while $f \circ \gamma = \gamma \circ f^\sharp$ means that no loss of precision is accumulated by approximating the result of computations on abstract objects. We follow [15] where the first is called *backward* ($\mathcal{B}$) and the second is called *forward* ($\mathcal{F}$) completeness. The problem of making abstract domains $\mathcal{B}$-complete has been solved in [17]. These results have been extended to $\mathcal{F}$-completeness in [15]. The key point in this construction is that there exists an either $\mathcal{B}$ or $\mathcal{F}$-complete abstract function $f^\sharp$ in an abstract domain A iff the best correct approximation $\alpha \circ f \circ \gamma$ of f in A is respectively either $\mathcal{B}$ or $\mathcal{F}$ complete. This means that both $\mathcal{F}$ and $\mathcal{B}$ completeness are properties of the underlying abstract domain A relatively to the concrete function f. In a more general setting let $f : C_1 \to C_2$ be a function on complete lattices C_1 and C_2, and $\rho \in uco(C_2)$ and $\eta \in uco(C_1)$ be abstract domains. $\langle \rho, \eta \rangle$ is a pair of $\mathcal{B}(\mathcal{F})$-complete abstractions for f if $\rho \circ f = \rho \circ f \circ \eta$ ($f \circ \eta = \rho \circ f \circ \eta$). In the following we denote by $\mathcal{F}(C_1, C_2, f) \stackrel{\text{def}}{=} \{\ \langle \rho, \eta \rangle \,|\, f \circ \eta = \rho \circ f \circ \eta\ \}$ and $\mathcal{B}(C_1, C_2, f) \stackrel{\text{def}}{=} \{\ \langle \rho, \eta \rangle \,|\, \rho \circ f = \rho \circ f \circ \eta\ \}$. A pair of domain transformers can be associated with any completeness problem. We follow [11, 16] by defining a *domain refinement* and *simplification* as any monotone function $\tau : uco(L) \to uco(L)$ such that $X \subseteq \tau(X)$ and $\tau(X) \subseteq X$ respectively. In [17] and [15], a constructive characterization of the most abstract refinement, called *complete shell*, and of the most concrete simplification, called *complete core*, of any domain, making it $\mathcal{F}$ or $\mathcal{B}$ complete, for a given continuous function f, is given as a solution of a simple domain equation. Consider the following basic operators on closures:

$R_f^{\mathcal{F}} \stackrel{\text{def}}{=} \lambda X.\, \mathcal{M}(f(X))$	$R_f^{\mathcal{B}} \stackrel{\text{def}}{=} \lambda X.\, \mathcal{M}(\bigcup_{y \in X} \max(f^{-1}(\downarrow y)))$
$C_f^{\mathcal{F}} \stackrel{\text{def}}{=} \lambda X.\, \{\ y \in L \,|\, f(y) \subseteq X\ \}$	$C_f^{\mathcal{B}} \stackrel{\text{def}}{=} \lambda X.\, \{\ y \in L \,|\, \max(f^{-1}(\downarrow y)) \subseteq X\ \}$

Let $\ell \in \{\mathcal{F}, \mathcal{B}\}$. In [17] the authors proved that the only interesting cases, as far as the refinement and simplification towards ℓ-completeness are concerned, are respectively the most concrete $\beta \sqsupseteq \rho$ such that $\langle \beta, \eta \rangle$ is ℓ-complete and the most abstract $\beta \sqsubseteq \eta$ such that $\langle \rho, \beta \rangle$ is ℓ-complete. In particular given $\rho \in uco(C_2)$ the ℓ-complete shell of $\eta \in uco(C_1)$ is $\mathcal{R}_f^{\ell,\rho}(\eta) \stackrel{\text{def}}{=} \eta \sqcap R_f^\ell(\rho)$ and given

$\eta \in uco(C_1)$ the ℓ-complete core of $\rho \in uco(C_2)$ is $\mathcal{C}_f^{\ell,\eta}(\rho) \stackrel{\text{def}}{=} \rho \sqcup C_f^\ell(\eta)$. Note that, when f is additive $\max\{\, x \mid f(x) \leq y \,\} = \bigvee\{\, x \mid f(x) \leq y \,\} = f^+$, and therefore $\mathcal{B}(C_1, C_2, f) = \mathcal{F}(C_2, C_1, f^+)$ (cf. [15]). Clearly, when we consider $f : C \to C$ and the constraint $\eta = \rho$, the above construction requires a fixpoint iteration on abstract domains: $\mathcal{R}_f^\ell(\rho) = gfp(\lambda X.\ \rho \sqcap R_f^\ell(X))$ and $\mathcal{C}_f^\ell(\rho) = lfp(\lambda X.\ \rho \sqcup C_f^\ell(X))$ are called respectively the *absolute ℓ-complete shell* and *core* of ρ for f. Note that $\mathcal{R}_f^\ell \in lco(uco(C))$ and $\mathcal{C}_f^\ell \in uco(uco(C))$ (see [17]). It is worth noting that ℓ-complete cores and shells are adjoint abstract domain transformers, i.e., adjoint functions on the lattice of abstract interpretations. For any $\eta \in uco(C_1)$ and $\rho \in uco(C_2)$: $C_f^\ell(\eta) \sqsubseteq \rho \ \Leftrightarrow\ \eta \sqsubseteq R_f^\ell(\rho)$, which, by definition, implies that $\mathcal{C}_f^{\ell,\eta}(\rho) \sqsubseteq \rho \ \Leftrightarrow\ \eta \sqsubseteq \mathcal{R}_f^{\ell,\rho}(\eta)$.

3 Information Flows in Language-Based Security

Confidential data are considered *private*, labeled with `H` (high-level of secrecy), while all other data are public, labeled with `L` (low-level of secrecy) [10]. Non-interference can be naturally expressed by using semantic models of program execution. This idea goes back to Cohen's work on *strong dependency* [6], which uses denotational semantics for modeling how information can be transmitted among variables during the execution of programs. Therefore non-interference for programs essentially means that *"a variation of confidential (high or private) input does not cause a variation of public (low) output"* [27]. When this happens, we say that the program has only *secure information flows* [1, 6, 9, 10, 19, 30]. This situation has been modeled by considering the denotational (input/output) semantics $[\![P]\!]$ of the program P. In particular we consider programs where data are typed as private (`H`) or public (`L`). Program states in Σ are functions (represented as tuples) mapping variables in the set of values $\mathbb{V}$. Finite traces on Σ are denoted Σ^+. If $\mathtt{T} \in \{\mathtt{H}, \mathtt{L}\}$, $n = |\{x \in Var(P) | x : \mathtt{T}\}|$, and $v \in \mathbb{V}^n$, we abuse notation by denoting $v \in \mathbb{V}^{\mathtt{T}}$ the fact that v is a possible value for the variables with security type `T`. Moreover, we assume that any input s, can be seen as a pair (h, l), where $s^{\mathtt{H}} = h$ is a value for private data and $s^{\mathtt{L}} = l$ is a value for public data. In this case, *non-interference* can be formulated as follows.

A program P is *secure* if
$\forall$ input $s, t\ .\ s^{\mathtt{L}} = t^{\mathtt{L}} \ \Rightarrow\ ([\![P]\!](s))^{\mathtt{L}} = ([\![P]\!](t))^{\mathtt{L}}$

This problem has been formulated also as a *Partial Equivalence Relation* (PER) [28]. In this case we have that if the input data are equivalent under a given equivalent relation, then also the outputs are equivalent w.r.t. a corresponding output equivalence relation. The result is a PER on the domain of semantic functions which can be used to model non-interference as above, where the equality on public data can be generalized by considering any equivalence relation. McLean [23] treats possibilistic notions of non-interference for even non-deterministic programs in the context of trace semantics: A program is secure if the set of its traces is closed under a function *purge*, i.e., it is insensible by varying private

inputs. In [22] the different notions of possibilistic non-interference are modeled in a odular way. Ryan [25], Focardi and Gorrieri [12] all provide a comprehensive treatment of non-interference for concurrent programs in process algebras, where attackers are modeled as view relations on computation trees. The standard methods for checking non-interference are based on security-type systems and data-flow/control-flow analysis. Type-based approaches are designed in such a way that well-typed programs do not leak secrets. In a security-typed language, a type is inductively associated at compile-time with program statements in such a way that any statement showing a potential flow disclosing secrets is rejected [29, 31]. Similarly, data-flow/control-flow analysis techniques are devoted to statically discover flows of secret data into public variables [3, 4, 19, 20, 28]. All these approaches are characterized by the way they model attackers (or unauthorized users).

3.1 Joshi and Leino's Semantic-Based Approach

As we said above, a program is secure if any observation of the initial and final values of $l : \mathtt{L}$ do not provide any information about the initial value of $h : \mathtt{H}$ [19]. Assume that the adversary has knowledge of the program text and of the initial and final values of l. The idea of Joshi and Leino's semantic-based approach to language-based security is that of characterizing secure information flow as program equivalence, denoted by $\doteq$. They introduce a program $\mathtt{HH} \stackrel{\text{def}}{=}$ "assign to h an arbitrary value". Consider a program P for which we want to prove non-interference. The program $\mathtt{HH}; P$ corresponds to run P after having set h to an arbitrary value; while the program $P; \mathtt{HH}$ discards the final value of h resulting from the execution of P. Then a program P is said to be *secure* if

$$\mathtt{HH}\ ;\ P;\ \mathtt{HH}\ \doteq\ P\ ;\ \mathtt{HH} \tag{1}$$

where $\doteq$ is the relational input/output semantic equality between programs, namely for each possible input the two programs have to show the same public output behavior. In order to understand this characterization, note that the occurence of HH after P on both the sides of the equality indicates that only the final values of l are of interest, whereas the occurence of HH before P on the left side of the equality indicates that the program starts with an arbitrary assignment to h. Clearly, the two programs are input/output equivalent provided that the final value of l, produced by P, does not depend on the initial value of h, which is indeed standard non-interference.

3.2 Robust Declassification

Declassifying information means downgrading the sensitivity of data in order to accommodate with (intentional) information leakage. Robust declassification has been introduced in [32] as a systematic method to drive declassification by characterizing what information flows from confidential to public variables. In particular the observational attacker's capability is modeled by using equivalence relations as in PER models, and declassification of private data is obtained by

Table 1. Narrow and Abstract Non-Interference

$[\eta]P(\rho)$ if $\forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}, \forall l_1, l_2 \in \mathbb{V}^{\mathtt{L}} \,.\, \eta(l_1) = \eta(l_2) \;\Rightarrow\; \rho([\![P]\!](h_1, l_1)^{\mathtt{L}}) = \rho([\![P]\!](h_2, l_2)^{\mathtt{L}})$
$(\eta)P(\phi \rightsquigarrow \rho)$ if $\forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}, \forall l \in \mathbb{V}^{\mathtt{L}} \,.\; \rho([\![P]\!](\phi(h_1), \eta(l))^{\mathtt{L}}) = \rho([\![P]\!](\phi(h_2), \eta(l))^{\mathtt{L}})$

manipulating these relations in a semantic-driven way. The semantics considered is the operational semantics, defined on a transition system. The authors provide a systematic method for identifying what the attacker could observe of the concrete execution traces, by iteratively refining the initial equivalence relation on the states of the program. At this point they declassify private data in order to make the attacker *blind*, i.e., they declassify all the information that the attacker can get from the execution of the program.

3.3 Abstract Non-interference: Attack Models and Declassification

In [13], we introduced the notion of abstract non-interference modeling weaker information flows, attack models, and declassification. The idea is that an attacker can observe only some properties, modeled as abstract interpretations of program semantics, of public concrete values. The *model of an attacker*, also called *attacker*, is therefore a pair of abstractions $\langle \eta, \rho \rangle$, with $\eta, \rho \in uco(\wp(\mathbb{V}^{\mathtt{L}}))$, representing what an observer can see about, respectively, the input and output of a program. The notion of *narrow (abstract) non-interference* (NNI) represents the first weakening of standard non-interference relatively to a given model of an attacker. When a program P satisfies narrow non-interference we write $[\eta]P(\rho)$, see Table 1. The problem with this notion is that it introduces *deceptive flows* [13]. Consider, for instance, $l := l * h^2$, and consider the public input property of being an even number, then we can observe a variation of the output's sign due to the existence of both negative and positive even numbers, revealing flows which does not depend on the private data, here called deceptive. In order to avoid deceptive interference we introduce a weaker notion of non-interference, having no deceptive flows, yet modeling properties of informations flows. Namely, such that, when the attacker is able to observe the property η of public input, and the property ρ of public output, then no information flow concerning the property ϕ of the private input is observable from the public output. Namely, ϕ represents the confidentialinformation that we want to keep secret. We call this notion *abstract non-interference* (ANI). When a program P satisfies abstract non-interference we write $(\eta)P(\phi \rightsquigarrow \rho)$, where $\phi \in uco(\wp(\mathbb{V}^{\mathtt{H}}))$, see Table 1. Note that $[\mathtt{id}]P(\mathtt{id})$ models exactly (standard) non-interference. Moreover, we have that abstract non-interference is a weakening of both, standard and narrow non-interference: $\forall \eta, \rho \in uco(\wp(\mathbb{V}^{\mathtt{L}})), \phi \in uco(\wp(\mathbb{V}^{\mathtt{H}}))$ we have $[\mathtt{id}]P(\mathtt{id}) \;\Rightarrow\; (\eta)P(\phi \rightsquigarrow \rho)$ and $[\eta]P(\rho) \;\Rightarrow\; (\eta)P(\phi \rightsquigarrow \rho)$, while standard non-interference is not stronger than the narrow one due to deceptive interference. A proof-system has been introduced, in [14], for checking both narrow and abstract non-interference inductively on program's syntax. Moreover, in [13], two methods for deriving the most concrete output observation for a program, given

the input one, for both narrow and abstract non-interference are provided. In particular the idea is that of collecting in the same abstract object all the elements that, if distinguished, would generate a visible flow. These most concrete output observations that are not able to get information from the program P, observing η in input, are, respectively, denoted $[\eta]\llbracket P\rrbracket(\mathtt{id})$ and $(\eta)\llbracket P\rrbracket(\phi \rightsquigarrow \mathtt{id})$, both in $uco(\wp(\mathbb{V}^{\mathtt{L}}))$. The following theorem is proved in [13].

Theorem 1. $[\eta]\llbracket P\rrbracket(\mathtt{id}) \sqsubseteq \rho \;\Leftrightarrow\; [\eta]P(\rho)$, $(\eta)\llbracket P\rrbracket(\phi \rightsquigarrow \mathtt{id}) \sqsubseteq \rho \;\Leftrightarrow\; (\eta)P(\phi \rightsquigarrow \rho)$.

Example 1. Consider the properties *Sign* and *Par*, observing, respectively, the sign and the parity of integers, and the program fragment: $P \stackrel{\text{def}}{=} l := l * h^2$. with security typing: $h : \mathtt{H}$ and $l : \mathtt{L}$ and $\mathbb{V} = \mathbb{Z}$. Let us check if $(\mathtt{id})P(\mathtt{id} \rightsquigarrow Par)$. Note that $Par(\llbracket P\rrbracket(2,1)^{\mathtt{L}}) = Par(4) = 2\mathbb{Z}$ while $Par(\llbracket P\rrbracket(3,1)^{\mathtt{L}}) = Par(9) = 2\mathbb{Z}+1$, which are clearly different, therefore in this case $(\mathtt{id})P(\mathtt{id} \rightsquigarrow Par)$ doesn't hold. Consider $(\mathtt{id})P(Sign \rightsquigarrow Par)$. Note that $Par(\llbracket P\rrbracket(Sign(2),1)^{\mathtt{L}}) = Par(\llbracket P\rrbracket(Sign(3),1)^{\mathtt{L}}) = Par(0+) = \mathbb{Z}$. In this case it is simple to check that $(\mathtt{id})P(Sign \rightsquigarrow Par)$ holds.

The PER model of non-interference can be easily viewed as a narrow non-interference, where both input and output closures are partitioning, i.e., equivalence relations. This corresponds to narrow non-interference because in PERs the equivalence is checked on the program outputs of concrete computations. Abstract non-interference provides also an abstraction of declassification. The idea is to find the most abstract property on confidential data which has to be declassified in order to guarantee secrecy. For this reason we define the set:

$$\Pi_{\mathtt{P}}(\eta,\rho) \stackrel{\text{def}}{=} \left\{ \langle \left\{ h \in \mathbb{V}^{\mathtt{H}} \,\middle|\, \rho(\llbracket P\rrbracket(\langle h, \eta(l)\rangle)^{\mathtt{L}}) = A \right\}, \eta(l)\rangle \,\middle|\, l \in \mathbb{V}^{\mathtt{L}},\ A \in \rho \right\}$$

This is the set of all the pairs $\langle H, L\rangle \in \wp(\mathbb{V}^{\mathtt{H}}) \times \wp(\mathbb{V}^{\mathtt{L}})$, such that, whenever $\eta(l) = L$, then for any $h_1, h_2 \in H$, no information flows, from private to public, are revealed. We use this set for deriving a partition of private data that guarantees secrecy. For each $L \in \eta$, we define $\Pi_{\mathtt{P}}(\eta,\rho)_{|L} \stackrel{\text{def}}{=} \left\{ H \,\middle|\, \langle H, L\rangle \in \Pi_{\mathtt{P}}(\eta,\rho) \right\}$. The partition on private data corresponds to the most abstract property that flows when the property observed of the public input is L: $\mathcal{P}(\bigcap_{L\in\eta} \mathcal{M}(\Pi_{\mathtt{P}}(\eta,\rho)_{|L}))$. In particular, it is the most abstract property that contains all the possible variations of private inputs that generate insecure information flows, and the most concrete such that each variation generates a flow. namely it uniquely represents the confidential information that flows into the public output.

Example 2. Consider the program fragment: $P = \quad l := l * h^2$. $\Pi_{\mathtt{P}}(\mathtt{id}, Par)$ is the set $\left\{ \langle \mathbb{Z}, l\rangle \,\middle|\, l \in 2\mathbb{Z} \right\} \cup \left\{ \langle 2\mathbb{Z}, l\rangle \,\middle|\, l \in 2\mathbb{Z}+1 \right\} \cup \left\{ \langle 2\mathbb{Z}+1, l\rangle \,\middle|\, l \in 2\mathbb{Z}+1 \right\}$. Therefore by using the notation above we have that if $l \in 2\mathbb{Z}$ then $\Pi_{\mathtt{P}}(\mathtt{id}, Par)_{|l} = \mathbb{Z}$ and if $l \in 2\mathbb{Z}+1$ then $\Pi_{\mathtt{P}}(\mathtt{id}, Par)_{|l} = \{2\mathbb{Z}, 2\mathbb{Z}+1\}$. Therefore, the most abstract partition on private data that can be declassified is $\{2\mathbb{Z}, 2\mathbb{Z}+1\}$. In other words we have that by looking at the low variables the only information that leaks about the high variables is its parity.

In order to adapt robust declassification in [32] to the abstract non-interference case, we consider passive attackers only and a semantics observing the initial and the final states of computations. We follow [32] in defining the information leaked by an equivalence relation transformer $S[\eta,\rho]$ on Σ for each

$\eta, \rho \in uco(\wp(\mathbb{V}^{\mathtt{L}}))$: $s_1 S[\eta,\rho] s_2$ iff $s_1^{\mathtt{L}} \approx_\eta s_2^{\mathtt{L}}$ and $(\forall \sigma, \delta \in \Sigma^+ . \sigma_\vdash = s_1 \wedge \delta_\vdash = s_2 \Rightarrow \sigma_\dashv^{\mathtt{L}} \approx_\rho \delta_\dashv^{\mathtt{L}}))$, where $s_1, s_2 \in \Sigma$ and given $\sigma \in \Sigma^+$ such that $|\sigma| = n \in \mathbb{N}$, then $\sigma_\vdash \stackrel{\text{def}}{=} \sigma_0$ and $\sigma_\dashv \stackrel{\text{def}}{=} \sigma_{n-1}$. It is simple to verify that $s_1 S[\eta,\rho] s_2$ iff $\eta(s_1^{\mathtt{L}}) = \eta(s_2^{\mathtt{L}})$ and $\rho([\![P]\!](s_1)^{\mathtt{L}}) = \rho([\![P]\!](s_2)^{\mathtt{L}})$. This means that abstract robust declassification *a la* [32] characterizes the information leaked in narrow abstract non-interference.

4 Abstract Non-interference as Completeness

Joshi and Leino's semantic-based approach to information flows [19] provides a way to interpret abstract non-interference as the problem of making an abstraction complete [17]. By considering the denotational semantics of a program P, $[\![P]\!]$, the Equation (1) becomes a *backward* completeness problem if the semantics of `HH` could be described as an abstraction. Indeed the program that associates with private variables an arbitrary value can be interpreted as the closure that abstracts the private value to the *"don't know"* abstract value, i.e., the set of all the possible values for private variables. Therefore, we define the function $\mathcal{H} : \wp(\mathbb{V}) \longrightarrow \wp(\mathbb{V})$ in the following way (recall that $\wp(\mathbb{V}) = \wp(\mathbb{V}^{\mathtt{H}} \times \mathbb{V}^{\mathtt{L}})$): $\mathcal{H} = \lambda X. \langle \mathbb{V}^{\mathtt{H}}, X^{\mathtt{L}} \rangle$, where $X^{\mathtt{L}} \stackrel{\text{def}}{=} \left\{ \, l \,\middle|\, \langle h, l \rangle \in X \, \right\}$. It is straightforward to prove its monotonicity, idepotence and extensivity. So we can finally conclude that

$$[\![\mathtt{HH}\ ;\ P;\ \mathtt{HH}]\!] = \mathcal{H} \circ [\![P]\!] \circ \mathcal{H} \text{ and } [\![P;\ \mathtt{HH}]\!] = \mathcal{H} \circ [\![P]\!]$$

Hence, non-interference can be equivalentely formalized as $\mathcal{H} \circ [\![P]\!] \circ \mathcal{H} = \mathcal{H} \circ [\![P]\!]$. The idea is to transform $\mathcal{H}$ in order to either refine or simplify the abstraction in order to get completeness, and therefore, abstract non-interference. This can be achieved by observing that $\mathcal{H} = \lambda X. \langle \mathbb{T}(X^{\mathtt{H}}), \mathtt{id}(X^{\mathtt{L}}) \rangle = \lambda X. \langle \mathbb{V}^{\mathtt{H}}, X^{\mathtt{L}} \rangle$ where $X^{\mathtt{H}} \stackrel{\text{def}}{=} \left\{ \, h \,\middle|\, \langle l, h \rangle \in X \, \right\}$, i.e., $\mathcal{H}$ is the product of respectively the top and the bottom abstractions in the lattice of abstract interpretations. This means that the private component of $\mathcal{H}$ can only be refined as well as we can only abstract its public one. In this context we prove that shell and core have two different and precise meanings: The core abstracts the public component, viz. characterizes the most concrete attacker that cannot disclose private properties; The shell refines the private component, viz. characterizes the most abstract property that flows.

Example 3. Consider $P \stackrel{\text{def}}{=} l := 2 * h$, where $l : \mathtt{L}$ and $h : \mathtt{H}$. P violates non-interference, e.g., $\mathcal{H} \circ [\![P]\!] \circ \mathcal{H}(\langle 2, 3 \rangle) = \mathcal{H} \circ [\![P]\!](\langle \mathbb{Z}, 3 \rangle) = \mathcal{H}(\langle \mathbb{Z}, 2\mathbb{Z} \rangle) = \langle \mathbb{Z}, 2\mathbb{Z} \rangle$ while $\mathcal{H} \circ [\![P]\!](\langle 2, 3 \rangle) = \mathcal{H}(\langle 2, 4 \rangle) = \langle \mathbb{Z}, 4 \rangle$, where $2\mathbb{Z} \neq 4$. We can derive the complete core of $\mathcal{H}$, which makes the program secure. From [17] we have to keep only those elements whose inverse image is a fix-point of $\mathcal{H}$: $\mathcal{C}^{\mathcal{B}}_{[\![P]\!]}(\mathcal{H}) = \left\{ \, \langle \mathbb{Z}, L \rangle \,\middle|\, \left\{ \, \langle h, l \rangle \,\middle|\, \langle h, 2h \rangle \subseteq \langle \mathbb{Z}, L \rangle \, \right\} \subseteq \mathcal{H} \, \right\}$. Note that $\langle H, L \rangle \in \mathcal{H}$ iff $H = \mathbb{Z}$ and $\langle \mathbb{Z}, L' \rangle \supseteq \langle \mathbb{Z}, 2\mathbb{Z} \rangle$ iff $L' \supseteq 2\mathbb{Z}$. More generally, if $L' \subseteq 2\mathbb{Z} + 1$ then $\langle h, 2h \rangle \in \langle \mathbb{Z}, L' \rangle$ is false for each possible L', namely $\left\{ \, \langle h, l \rangle \,\middle|\, [\![P]\!](\langle h, l \rangle) \subseteq \langle H', L' \rangle \, \right\} = \varnothing \subseteq \mathcal{H}$ which means that in this case $\langle \mathbb{Z}, L' \rangle$ is kept. Therefore, $\mathcal{C}^{\mathcal{B}}_{[\![P]\!]}(\mathcal{H}) = \left\{ \, \langle \mathbb{Z}, L \rangle \,\middle|\, L \cap 2\mathbb{Z} \in \{2\mathbb{Z}, \varnothing\} \, \right\}$, which corresponds to abstracting the public output in the domain that is not able to distinguish even numbers. Let $\overline{\mathcal{H}} \stackrel{\text{def}}{=} \mathcal{C}^{\mathcal{B}}_{[\![P]\!]}(\mathcal{H})$, then in the previous case, we have $\overline{\mathcal{H}} \circ [\![P]\!] \circ \overline{\mathcal{H}}(\langle 2, 3 \rangle) = \overline{\mathcal{H}} \circ [\![P]\!](\langle \mathbb{Z}, 3 \rangle) = \overline{\mathcal{H}}(\langle \mathbb{Z}, 2\mathbb{Z} \rangle) = \langle \mathbb{Z}, 2\mathbb{Z} \rangle$ and $\overline{\mathcal{H}} \circ [\![P]\!](\langle 2, 3 \rangle) = \overline{\mathcal{H}}(\langle 2, 4 \rangle) = \langle \mathbb{Z}, 2\mathbb{Z} \rangle$.

Example 4. Consider $P \stackrel{\text{def}}{=} l := (2h+1) \bmod 2$, where $l : \mathtt{L}$ and $h : \mathtt{H}$. The program violates non-interference, since, for instance, $\mathcal{H}\circ[\![P]\!]\circ\mathcal{H}(\langle 2,3\rangle) = \mathcal{H}\circ[\![P]\!](\langle\mathbb{Z},3\rangle) = \mathcal{H}(\langle\mathbb{Z},\{-1,1\}\rangle) = \langle\mathbb{Z},\{-1,1\}\rangle$ while $\mathcal{H}\circ[\![P]\!](\langle 2,3\rangle) = \mathcal{H}(\langle 2,1\rangle) = \langle\mathbb{Z},1\rangle$ and $\{-1,1\} \neq 1$. We compute the complete shell of $\mathcal{H}$, characterizing the flowing property of private information, namely we add all the inverse images of the elements in $\mathcal{H}$.

$$\mathcal{R}^{\mathcal{B}}_{[\![P]\!]}(\mathcal{H}) = \mathcal{H} \sqcap \mathcal{M}(\bigcup_{L' \in \wp(\mathbb{V}^{\mathtt{L}})} \left\{ \langle h,l\rangle \,\middle|\, \langle h, 2h+1 \bmod 2\rangle \in \langle\mathbb{Z},L'\rangle \right\})$$

If $-1 \notin L'$, then $\left\{ \langle h,l\rangle \,\middle|\, \langle h, 2h+1 \bmod 2\rangle \in \langle\mathbb{Z},L'\rangle \right\} = \langle\mathbb{Z}_0^+,\mathbb{Z}\rangle$, $\mathbb{Z}_0^+ \stackrel{\text{def}}{=} \mathbb{Z}^+ \cup \{0\}$. If $1 \notin L'$, then $\left\{ \langle h,l\rangle \,\middle|\, \langle h, 2h+1 \bmod 2\rangle \in \langle\mathbb{Z},L'\rangle \right\} = \langle\mathbb{Z}^-,\mathbb{Z}\rangle$. Finally, if $1,-1 \notin L'$ we have $\left\{ \langle h,l\rangle \,\middle|\, \langle h, 2h+1 \bmod 2\rangle \in \langle\mathbb{Z},L'\rangle \right\} = \varnothing$.
Hence $\bigcup_{L' \in \wp(\mathbb{V}^{\mathtt{L}})} \left\{ \langle h,l\rangle \,\middle|\, \langle h, 2h+1 \bmod 2\rangle \in \langle\mathbb{Z},L'\rangle \right\} = \{\langle\mathbb{Z}_0^+,\mathbb{Z}\rangle, \langle\mathbb{Z}^-,\mathbb{Z}\rangle, \varnothing\}$, which implies $\mathcal{R}^{\mathcal{B}}_{[\![P]\!]}(\mathcal{H}) = \mathcal{H} \cup \left\{ \langle H,L\rangle \,\middle|\, H \in \{\mathbb{Z}_0^+,\mathbb{Z}^-\},\ L \in \wp(\mathbb{V}^{\mathtt{L}}) \right\}$. Let $\overline{\mathcal{H}} \stackrel{\text{def}}{=} \mathcal{R}^{\mathcal{B}}_{[\![P]\!]}(\mathcal{H})$, then $\overline{\mathcal{H}}\circ[\![P]\!]\circ\overline{\mathcal{H}}(\langle 2,3\rangle) = \overline{\mathcal{H}}\circ[\![P]\!](\langle\mathbb{Z}_0^+,3\rangle) = \overline{\mathcal{H}}(\langle\mathbb{Z}_0^+,\{1\}\rangle) = \langle\mathbb{Z}_0^+,\{1\}\rangle$ and $\overline{\mathcal{H}}\circ[\![P]\!](\langle 2,3\rangle) = \overline{\mathcal{H}}(\langle 2,1\rangle) = \langle\mathbb{Z}_0^+,\{1\}\rangle$.

The idea is to embed the model of an attacker as given in ANI, i.e., as a pair of input/output abstractions, in $\mathcal{H}$. Consider $\langle\wp(\mathbb{V}^{\mathtt{H}}) \times \wp(\mathbb{V}^{\mathtt{L}}), \varnothing, \langle\mathbb{V}^{\mathtt{H}},\mathbb{V}^{\mathtt{L}}\rangle, \Cup, \cap, \subseteq\rangle$, where $\langle H_1,L_1\rangle \Cup \langle H_2,L_2\rangle \stackrel{\text{def}}{=} \langle H_1 \cup H_2, L_1 \cup L_2\rangle$. It is well known that there exists an obvious GI of $\wp(\mathbb{V}^{\mathtt{H}}) \times \wp(\mathbb{V}^{\mathtt{L}})$ in $\wp(\mathbb{V}^{\mathtt{H}} \times \mathbb{V}^{\mathtt{L}})$, corresponding to the closure: $\mathtt{Split} \stackrel{\text{def}}{=} \lambda X.\ \left\{ \langle x_1,x_2\rangle \,\middle|\, \exists y.\ \langle x_1,y\rangle \in X,\ \exists z.\ \langle z,x_2\rangle \in X \right\}$. Consider the closure $\rho \in uco(\wp(\mathbb{V}^{\mathtt{L}}))$. We define $\mathcal{H}_\rho \in uco(\wp(\mathbb{V}^{\mathtt{H}}) \times \wp(\mathbb{V}^{\mathtt{L}}))$:

$$\mathcal{H}_\rho \stackrel{\text{def}}{=} \lambda X.\ \langle\mathbb{V}^{\mathtt{H}}, \rho(X^{\mathtt{L}})\rangle$$

Note that $\mathcal{H} = \mathcal{H}_{\mathtt{id}}$, $\mathcal{H}_\rho \in uco(\wp(\mathbb{V}^{\mathtt{H}} \times \mathbb{V}^{\mathtt{L}}))$, and for any pair of disjunctive closures $\eta,\rho \in uco(\mathbb{V}^{\mathtt{L}})$ and for all $\langle h,l\rangle \in \mathbb{V}$: $\mathcal{H}_\rho\circ[\![P]\!]\circ\mathcal{H}_\eta(\langle h,l\rangle) = \mathcal{H}_\rho\circ[\![P]\!](\langle h,l\rangle) \ \Leftrightarrow\ \mathcal{H}_\rho\circ[\![P]\!]\circ\mathcal{H}_\eta = \mathcal{H}_\rho\circ[\![P]\!]$.

Theorem 2. *Let $\rho,\eta \in uco(\wp(\mathbb{V}^{\mathtt{L}}))$.*

1. *$[\eta]P(\rho) \ \Leftarrow\ \mathcal{H}_\rho\circ[\![P]\!]\circ\mathcal{H}_\eta = \mathcal{H}_\rho\circ[\![P]\!]$;*
2. *If ρ is disjunctive and η is partitioning: $[\eta]P(\rho) \ \Rightarrow\ \mathcal{H}_\rho\circ[\![P]\!]\circ\mathcal{H}_\eta = \mathcal{H}_\rho\circ[\![P]\!]$.*

This result proves that narrow abstract non-interference is weaker than the generalization of Joshi and Leino's semantics-based approach to non-interference, which is a problem of completeness. Moreover, when both η and ρ are equivalence relations on public data as in the PER model, i.e., partitioning closures, then the narrow abstract non-interference is equivalent to the PER model of non-interference, which is in turn an instance of a completeness problem. Theorem 2 gives a slightly weaker condition, because all partitioning closures (viz. equivalence relations) are disjunctive, but the converse does not hold in general. In order to extend Theorem 2 to model abstract non-interference we have to modify the program semantics. The idea is to consider an abstract semantics that is applied to abstract (public and private) data. Consider the closures $\eta \in uco(\wp(\mathbb{V}^{\mathtt{L}}))$ and $\phi \in uco(\wp(\mathbb{V}^{\mathtt{H}}))$. We define the abstract semantics as $[\![P]\!]^{\eta,\phi} \stackrel{\text{def}}{=} \lambda\langle h,l\rangle.\ [\![P]\!](\phi(h),\eta(l))$. Note that, for any pair of disjunctive closures $\eta,\rho \in uco(\mathbb{V}^{\mathtt{L}})$ and for all $\langle h,l\rangle \in \mathbb{V}$: $\mathcal{H}_\rho\circ[\![P]\!]^{\eta,\phi}\circ\mathcal{H}_\eta(\langle h,l\rangle) = \mathcal{H}_\rho\circ[\![P]\!]^{\eta,\phi}(\langle h,l\rangle) \ \Leftrightarrow\ \mathcal{H}_\rho\circ[\![P]\!]^{\eta,\phi}\circ\mathcal{H}_\eta = \mathcal{H}_\rho\circ[\![P]\!]^{\eta,\phi}$.

Theorem 3. *Consider* $\eta, \rho \in uco(\wp(\mathbb{V}^{\mathtt{L}}))$ *and* $\phi \in uco(\wp(\mathbb{V}^{\mathtt{H}}))$*:*

1. $(\rho)P(\phi \rightsquigarrow \eta) \Leftarrow \mathcal{H}_\rho \circ [\![P]\!]^{\eta,\phi} \circ \mathcal{H}_\eta = \mathcal{H}_\rho \circ [\![P]\!]^{\eta,\phi}$;
2. *If* ρ *and* η *are disjunctive then:* $(\eta)P(\phi \rightsquigarrow \rho) \Rightarrow \mathcal{H}_\rho \circ [\![P]\!]^{\eta,\phi} \circ \mathcal{H}_\eta = \mathcal{H}_\rho \circ [\![P]\!]^{\eta,\phi}$.

Once again abstract non-interference is weaker than the generalization of Joshi and Leino's approach to non-interference. Additivity is here sufficient in order to let these two approaches equivalent. Because the only difference in the corresponding completeness problems is due to the program semantics: $[\![P]\!]^{\eta,\phi}$ in the case of abstract non-interference and $[\![P]\!]^{\mathtt{id},\mathtt{id}}$ in the narrow case, in the following, without loss of generality we consider the abstract non-interference case being more general. In the following we omit the apex $\mathcal{B}$ from shells and cores, since we will consider always backward completeness.

5 The Most Concrete *Observer* as Completeness Core

In [13] we gave a method for systematically deriving the most concrete harmless attacker (canonical attacker) associated with a given program. By Theorem 3, the most concrete public observer, which is the canonical attacker, can be derived as the most concrete abstraction satisfying the following completeness problem:

$$\mathcal{H} \circ [\![P]\!]^{\eta,\phi} \circ \mathcal{H}_\eta = \mathcal{H} \circ [\![P]\!]^{\eta,\phi} \tag{2}$$

Then we have the following result which allows us to specify the canonical attacker as the fix-point of an abstract domain simplification.

Theorem 4. *Let* $\eta \in uco(\wp(\mathbb{V}^{\mathtt{L}}))$ *be disjunctive and* $\phi \in uco(\mathbb{V}^{\mathtt{H}})$*. Then we have* $\mathcal{C}^{\mathcal{H}_\eta}_{[\![P]\!]^{\eta,\phi}}(\mathcal{H}) = \left\{ \langle \mathbb{V}^{\mathtt{H}}, L\rangle \,\middle|\, \left\{ \langle h, l\rangle \,\middle|\, [\![P]\!](\langle \phi(h), \eta(l)\rangle) \subseteq \langle \mathbb{V}^{\mathtt{H}}, L\rangle \right\} \in \mathcal{H}_\eta \right\}$ *and*

$$\left\{ L \in \wp(\mathbb{V}^{\mathtt{L}}) \,\middle|\, \langle \mathbb{V}^{\mathtt{H}}, L\rangle \in \mathcal{C}^{\mathcal{H}_\eta}_{[\![P]\!]^{\eta,\phi}}(\mathcal{H}) \right\} = (\eta)[\![P]\!](\phi \rightsquigarrow \mathtt{id}).$$

Example 5. Consider the following program fragment, with $l : \mathtt{L}$ and $h : \mathtt{H}$.

$$P \stackrel{\text{def}}{=} \textbf{while } h \textbf{ do } l := 2l;\ h := 0 \textbf{ endw} \qquad [\![P]\!](\langle h, l\rangle) = \begin{cases} \langle h, l\rangle & \text{if } h = 0 \\ \langle h, 2l\rangle & \text{otherwise} \end{cases}$$

We look for the core in order to make $\langle \mathcal{H}, \mathcal{H}\rangle$ complete for the map $[\![P]\!]^{\mathtt{id},\mathtt{id}} = [\![P]\!]$.

$$\mathcal{C}^{\mathcal{H}}_{[\![P]\!]}(\mathcal{H}) = \left\{ \langle \mathbb{Z}, L\rangle \,\middle|\, \forall l \in \mathbb{V}^{\mathtt{L}} \,.\, l \in L \Leftrightarrow 2l \in L \right\}$$

It is straightforward to show that $\mathcal{C}^{\mathcal{H}}_{[\![P]\!]}(\mathcal{H})$ is the domain that abstracts the public data in the domain $\curlyvee(\left\{ n\{2\}^{\mathbb{N}} \,\middle|\, n \in 2\mathbb{Z}+1 \right\})$, where $\{2\}^{\mathbb{N}} \stackrel{\text{def}}{=} \left\{ 2^k \,\middle|\, k \in \mathbb{N} \right\}$. Let $\overline{\mathcal{H}} \stackrel{\text{def}}{=} \mathcal{C}^{\mathcal{H}}_{[\![P]\!]}(\mathcal{H})$, then, $\overline{\mathcal{H}} \circ [\![P]\!] \circ \mathcal{H}(\langle 3, 5\rangle) = \overline{\mathcal{H}} \circ [\![P]\!](\mathbb{Z}, 5) = \overline{\mathcal{H}}(\langle \mathbb{Z}, \{5, 10\}\rangle) = \langle \mathbb{Z}, 5\{2\}^{\mathbb{N}}\rangle$, and $\overline{\mathcal{H}} \circ [\![P]\!](\langle 3, 5\rangle) = \overline{\mathcal{H}}(\langle \mathbb{Z}, \{10\}\rangle) = \langle \mathbb{Z}, 5\{2\}^{\mathbb{N}}\rangle$ while we have that $\mathcal{H} \circ [\![P]\!] \circ \mathcal{H}(\langle 3, 5\rangle) = \mathcal{H} \circ [\![P]\!](\mathbb{Z}, 5) = \mathcal{H}(\langle \mathbb{Z}, \{5, 10\}\rangle) = \langle \mathbb{Z}, \{5, 10\}\rangle$ and, on the other hand, $\mathcal{H} \circ [\![P]\!](\langle 3, 5\rangle) = \mathcal{H}(\langle \mathbb{Z}, \{10\}\rangle) = \langle \mathbb{Z}, \{10\}\rangle$.

6 The Most Abstract *Observable* as Completeness Shell

We are now interested in applying the same construction for characterizing the most abstract private observable, used for defining abstract declassification as a solution of a completeness problem in abstract interpretation. Namely, we are interested in the most abstract property that can be declassified in order to guarantee abstract non-interference. By Theorem 3, this information can be obtained by solving the following completeness problem:

$$\mathcal{H}_\rho \circ [\![P]\!]^{\eta,\mathtt{id}} \circ \mathcal{H}_\eta = \mathcal{H}_\rho \circ [\![P]\!]^{\eta,\mathtt{id}} \tag{3}$$

Lemma 1. *Let $\rho, \eta \in uco(\wp(\mathbb{V}^{\mathtt{L}}))$. Then we have*

$$\mathcal{R}^{\mathcal{H}_\rho}_{[\![P]\!]^{\eta,\mathtt{id}}}(\mathcal{H}_\eta) = \mathcal{H}_\eta \sqcap \mathcal{M}(\{\ \{\ \langle h, l\rangle \mid \rho([\![P]\!](\langle h, \eta(l)\rangle)^{\mathtt{L}}) \subseteq L\ \} \mid L \in \rho\ \}).$$

Moreover, let $\mathcal{R} \stackrel{def}{=} \mathcal{R}^{\mathcal{H}_\rho}_{[\![P]\!]^{\eta,\mathtt{id}}}(\mathcal{H}_\eta)$, then forall $l, l' \in \mathbb{V}^{\mathtt{L}}, h, h' \in \mathbb{V}^{\mathtt{H}}$ we have

$$\mathcal{R}(\langle h, l\rangle) = \mathcal{R}(\langle h', l'\rangle) \qquad \textit{iff} \qquad \rho[\![P]\!](\langle h, \eta(l)\rangle)^{\mathtt{L}} = \rho[\![P]\!](\langle h', \eta(l')\rangle)^{\mathtt{L}}$$

It is worth noting that, by Lemma 1, the partition induced by the complete shell of $\mathcal{H}_\eta$ on $\wp(\mathbb{V}^{\mathtt{H}} \times \mathbb{V}^{\mathtt{L}})$ for Equation 3 does not affect the closure η. This means that the only component which is actually refined is the abstraction on private data, and this corresponds to the most abstract partitioning of private data which can be declassified. This means that any change between equivalent elements does not produce insecure flows, as stated in the following theorem.

Theorem 5. *Let $\rho, \eta \in uco(\wp(\mathbb{V}^{\mathtt{L}}))$ then for each $l, l' \in \mathbb{V}^{\mathtt{L}}, h, h' \in \mathbb{V}^{\mathtt{H}}$ we have $\eta(l) = \eta(l') = Y \ \Rightarrow\ (\mathcal{R}(\langle h, l\rangle) = \mathcal{R}(\langle h', l'\rangle)$ iff $h' \in [h]_{\Pi_{\mathtt{P}}(\eta,\rho)_{|Y}})$.*

Next examples show how declassification can be obtained as solutions of completeness problems.

Example 6. Consider the program fragment: $P \stackrel{\text{def}}{=}\ l := l * h^2$, with $l : \mathtt{L}$ and $h : \mathtt{H}$. We want to find the shell in order to make $\langle \mathcal{H}, \mathcal{H}_{Par}\rangle$ complete for the map $[\![P]\!]^{\mathtt{id},\mathtt{id}} = [\![P]\!]$.

$$\mathcal{R}^{\mathcal{H}_{Par}}_{[\![P]\!]}(\mathcal{H}) = \mathcal{H} \sqcap \left(\left\{ \begin{array}{l} \langle \mathbb{Z}, \mathbb{Z}\rangle, \langle \mathbb{Z}, 2\mathbb{Z}\rangle \cup \langle 2\mathbb{Z}, 2\mathbb{Z}+1\rangle, \langle 2\mathbb{Z}+1, 2\mathbb{Z}+1\rangle, \\ \langle 2\mathbb{Z}+1, 2\mathbb{Z}\rangle, \varnothing \end{array} \right\} \right)$$

This means that the reduced product generates also $\langle 2\mathbb{Z}, 2\mathbb{Z}+1\rangle$ and therefore $\langle 2\mathbb{Z}, l\rangle$ for each $l \in 2\mathbb{Z}+1$. Let $\overline{\mathcal{H}} \stackrel{\text{def}}{=} \mathcal{R}^{\mathcal{H}_{Par}}_{[\![P]\!]}(\mathcal{H})$, then for instance, we have $\mathcal{H}_{Par} \circ [\![P]\!] \circ \overline{\mathcal{H}}(\langle 2, 3\rangle) = \mathcal{H}_{Par} \circ [\![P]\!](\langle 2\mathbb{Z}, 3\rangle) = \langle \mathbb{Z}, 2\mathbb{Z}\rangle$, and $\mathcal{H}_{Par} \circ [\![P]\!](\langle 2, 3\rangle) = \langle \mathbb{Z}, 2\mathbb{Z}\rangle$, while $\mathcal{H}_{Par} \circ [\![P]\!] \circ \mathcal{H}(\langle 2, 3\rangle) = \mathcal{H}_{Par} \circ [\![P]\!](\mathbb{Z}, 3) = \langle \mathbb{Z}, \mathbb{Z}\rangle$. As in abstract desclassification [13], this means that it is the variation of parity of the private input that generates the flow.

Example 7. Consider $\rho \stackrel{\text{def}}{=} \{\mathbb{Z}, 2\mathbb{Z}, 4\mathbb{Z}, 2\mathbb{Z}+1, \varnothing\}$ and $\eta \stackrel{\text{def}}{=} \{\mathbb{Z}, 2\mathbb{Z}, 5\mathbb{Z}, 10\mathbb{Z}, \varnothing\}$, and consider $P \stackrel{\text{def}}{=}$ **if** $(h \bmod 4) = 0$ **then** $l := l * h$ **else** $l := l * (h + 1)$ **fi**. Compute, first, the abstract robust declassification, as was introduced in [13]:

$$\Pi_{\mathtt{P}}(\eta, \rho) = \left\{ \begin{array}{l} \langle 4\mathbb{Z} \cup 4\mathbb{Z}+3, 10\mathbb{Z}\rangle, \langle 4\mathbb{Z}+1 \cup 4\mathbb{Z}+2, 10\mathbb{Z}\rangle, \\ \langle 4\mathbb{Z} \cup 4\mathbb{Z}+3, 5\mathbb{Z}\rangle, \langle 4\mathbb{Z}+1, 5\mathbb{Z}\rangle, \langle 4\mathbb{Z}+2, 5\mathbb{Z}\rangle, \\ \langle 4\mathbb{Z} \cup 4\mathbb{Z}+3, 2\mathbb{Z}\rangle, \langle 4\mathbb{Z}+1 \cup 4\mathbb{Z}+2, 2\mathbb{Z}\rangle, \\ \langle 4\mathbb{Z} \cup 4\mathbb{Z}+3, \mathbb{Z}\rangle, \langle 4\mathbb{Z}+1, \mathbb{Z}\rangle, \langle 4\mathbb{Z}+2, \mathbb{Z}\rangle \end{array} \right\}$$

Therefore we obtain $\Pi_{\mathtt{P}}(\eta, \rho)_{|10\mathbb{Z}} = \Pi_{\mathtt{P}}(\eta, \rho)_{|2\mathbb{Z}} = \{4\mathbb{Z} \cup 4\mathbb{Z}+3, 4\mathbb{Z}+1 \cup 4\mathbb{Z}+2\}$ and $\Pi_{\mathtt{P}}(\eta, \rho)_{|5\mathbb{Z}} = \Pi_{\mathtt{P}}(\eta, \rho)_{|\mathbb{Z}} = \{4\mathbb{Z} \cup 4\mathbb{Z}+3, 4\mathbb{Z}+1, 4\mathbb{Z}+2\}$. Consider now the completeness shell:

$$\begin{array}{r} \left\{ \langle h, l\rangle \,\middle|\, [\![P]\!](\langle h, \eta(l)\rangle)^{\mathtt{L}} \subseteq 4\mathbb{Z} \right\} = \langle 4\mathbb{Z} \cup 4\mathbb{Z}+3, \mathbb{Z}\rangle \\ \left\{ \langle h, l\rangle \,\middle|\, [\![P]\!](\langle h, \eta(l)\rangle)^{\mathtt{L}} \subseteq 2\mathbb{Z} \right\} = \langle \mathbb{Z} \smallsetminus 4\mathbb{Z}+2, \mathbb{Z}\rangle \cup \langle 4\mathbb{Z}+2, 2\mathbb{Z}\rangle \\ \left\{ \langle h, l\rangle \,\middle|\, [\![P]\!](\langle h, \eta(l)\rangle)^{\mathtt{L}} \subseteq 2\mathbb{Z}+1 \right\} = \varnothing \end{array}$$

Then we have:

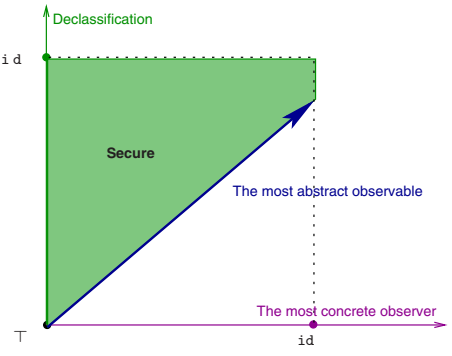
$$\mathcal{R}^{\mathcal{H}_\rho}_{[\![P]\!]}(\mathcal{H}_\eta) = \mathcal{H}_\eta \cup \left\{ \begin{array}{l} \langle \mathbb{Z} \smallsetminus 4\mathbb{Z}+2, \mathbb{Z}\rangle \cup \langle 4\mathbb{Z}+2, 2\mathbb{Z}\rangle, \langle 4\mathbb{Z} \cup 4\mathbb{Z}+3, \mathbb{Z}\rangle, \\ \langle \mathbb{Z} \smallsetminus 4\mathbb{Z}+2, 5\mathbb{Z}\rangle \cup \langle 4\mathbb{Z}+2, 10\mathbb{Z}\rangle, \langle 4\mathbb{Z} \cup 4\mathbb{Z}+3, 5\mathbb{Z}\rangle, \\ \langle 4\mathbb{Z} \cup 4\mathbb{Z}+3, 2\mathbb{Z}\rangle, \langle 4\mathbb{Z} \cup 4\mathbb{Z}+3, 10\mathbb{Z}\rangle \end{array} \right\}$$

For instance, consider $5, 9 \in 4\mathbb{Z}+1$, $6, 10 \in 4\mathbb{Z}+2$, and note that $\eta(10) = \eta(30) = 10\mathbb{Z}$, and $\eta(5) = \eta(15) = 5\mathbb{Z}$. Note that, 5 and 6 are in the same equivalence class in the partition induced by $\Pi_{\mathtt{P}}(\eta, \rho)_{|10\mathbb{Z}}$, written $5 \in [6]_{10\mathbb{Z}}$, and indeed $\mathcal{R}(\langle 5, 10\rangle) = \mathcal{R}(\langle 6, 30\rangle) = \langle \mathbb{Z}, 10\mathbb{Z}\rangle \in \mathcal{H}_\eta$. While $5 \in [9]_{5\mathbb{Z}} \neq [6]_{5\mathbb{Z}}$, namely the partition induced by $\Pi_{\mathtt{P}}(\eta, \rho)_{|5\mathbb{Z}}$ distinguishes 5 and 6, while 5 is together with 9 and 6 is together with 10, i.e., $10 \in [6]_{5\mathbb{Z}}$. On the other hand, we have $\mathcal{R}(\langle 5, 5\rangle) = \mathcal{R}(\langle 9, 15\rangle) = \langle \mathbb{Z} \smallsetminus 4\mathbb{Z}+2, 5\mathbb{Z}\rangle \cup \langle 4\mathbb{Z}+2, 10\mathbb{Z}\rangle$ and $\mathcal{R}(\langle 6, 5\rangle) = \mathcal{R}(\langle 10, 15\rangle) = \langle \mathbb{Z}, 5\mathbb{Z}\rangle \in \mathcal{H}_\eta$.

7 Adjoining Observer and Observable Properties

Modeling attackers means characterizing the maximal power of an harmless attacker, i.e., an attacker which cannot disclose confidential information. Declassification, instead, means characterizing the information revealed to a fixed attacker. As we have seen in the previous sections, the model of the most concrete harmless attacker corresponds to the most concrete public observer, while abstract declassification is characterized by the most abstract private observable. Clearly there is a strong relation between these two notions, since the more powerful is the attacker and the less is the confidential information that can be kept

private. In other words the index of the partition of private data for declassification is proportional to the cardinality of the abstract domain which models the precision of the property that the attacker can observe. This phenomenon can be precisely characterized in the lattice of abstract interpretations as an adjunction. In the picture on the right we provide a graphical representation of the relation existing between the most concrete property modeling the public observer and the most abstract property modeling the private observable. In particular, this picture represents the fact that the more powerful is the attacker, i.e., the more concrete is the observer property, the less confidential information can be kept private, i.e., the more concrete is the private observable. The picture also shows that, if the arrow represents the most abstract private observable, then when we declassify a confidential property which lays in the white area we cannot guarantee the secrecy of the program, since we are declassifying less than what is released by the semantics. When we declassify a property in the filled area instead, then we guarantee that no confidential information leakage may happen. Moreover, note that, even if the attacker is able to observe the value of public variables, then the observable property can be more abstract then the identity since the program itself can behave as a firewall for certain confidential properties, such as the square operation hides the sign. In Section 5 and 6 we proved that both problems can be viewed as instances of the problem of making abstractions complete. While the private observable for declassification is obtained by computing the completeness shell, the public observer, modeling the attacker, is obtained by computing the completeness core in the same completeness problem. These abstract domain transformers have been proved in [17] to be adjoint functions (see Sect. 2) on the lattice of abstract interpretations. The following result is therefore a consequence of Theorem 4 and 5.



Theorem 6. *Let* $\eta \in uco(\wp(\mathbb{V}^{\mathtt{L}}))$ *be a disjunctive property, and* P *a program. Then we have that* $\mathtt{id} \sqsubset (\eta)[\![P]\!](\mathtt{id} \rightsquigarrow \mathtt{id}) \Leftrightarrow \mathcal{P}(\sqcap_{L\in\eta}\mathcal{M}(\Pi_{\mathtt{P}}(\eta, \mathtt{id})_{|L})) \sqsubset \mathbb{T}$.

This result provides a precise mathematical framework where declassification and attack models can be systematically derived and compared with each other in the lattice of abstract interpretations by applying well known methods for abstract domain design. This framework can be the basis for applying quantitative methods and metrics [5] for measuring the amount of information leaked relatively to a given attack model, or by adjunction, the precision of an attacker under the hypothesis that some information can be declassified. Recently, several papers treated the problem of defining non-interference for programs where confidential information can be explicitly declassified [26, 21]. In these cases the authors define weaker notions of non-interference in order to model confinement

problem for programs where there are intentional releases of information. Abstract non-interference, instead does not consider explicit declassification, but allows to characterize which confidential information should be declassified, at least, in order to guarantee only secure information flows.

References

1. D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE Corp. Badford, MA, 1973.
2. T.S. Blyth and M.F. Janowitz. *Residuation theory*. Pergamon Press, 1972.
3. C. Bodei, P. Degano, F. Nielson, and H.R. Nielson. Static analysis for secrecy and non-interference in networks of processes. In *Proc. of PaCT'01*, volume 2127 of *Lecture Notes in Computer Science*, pages 27–41. Springer-Verlag, 2001.
4. D. Clark, C. Hankin, and S. Hunt. Information flow for algol-like languages. *Computer Languages*, 28(1):3–28, 2002.
5. D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. In *Workshop on Quantitative Aspects of Programming Laguages (QAPL '01)*, volume 59 of *Electronic Notes in Theoretical Computer Science*. Elsevier, Amsterdam, 2001.
6. E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating System Review*, 11(5):133–139, 1977.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of Conf. Record of the 4th ACM Symp. on Principles of Programming Languages* (*POPL '77*), pages 238–252. ACM Press, New York, 1977.
8. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of Conf. Record of the 6th ACM Symp. on Principles of Programming Languages* (*POPL '79*), pages 269–282. ACM Press, New York, 1979.
9. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.
10. D. E. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
11. G. Filé, R. Giacobazzi, and F. Ranzato. A unifying view of abstract domain design. *ACM Comput. Surv.*, 28(2):333–336, 1996.
12. R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer security*, 3(1):5–33, 1995.
13. R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*, pages 186–197. ACM-Press, NY, 2004.
14. R. Giacobazzi and I. Mastroeni. Proving abstract non-interference. In *Annual Conference of the European Association for Computer Science Logic (CSL'04)*, volume 3210, pages 280–294. Springer-Verlag, 2004.
15. R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model-checking. In P. Cousot, editor, *Proc. of The 8th Internat. Static Analysis Symp. (SAS'01)*, volume 2126 of *Lecture Notes in Computer Science*, pages 356–373. Springer-Verlag, 2001.

16. R. Giacobazzi and F. Ranzato. Refining and compressing abstract domains. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proc. of the 24th Internat. Colloq. on Automata, Languages and Programming* (*ICALP '97*), volume 1256 of *Lecture Notes in Computer Science*, pages 771–781. Springer-Verlag, Berlin, 1997.
17. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. of the ACM.*, 47(2):361–416, 2000.
18. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.
19. R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37:113–138, 2000.
20. P. Laud. Semantics and program analysis of computationally secure information flow. In *Programming Languages and Systems, 10th European Symp. On Programming, ESOP*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag, 2001.
21. P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. of the 32st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM-Press, NY, 2005. To appear.
22. H. Mantel. Possibilistic definitions of security – an assemply kit –. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 185–199. IEEE Computer Society Press, 2000.
23. J. McLean. Proving noninterference and functional correcteness using traces. *Journal of Computer security*, 1(1):37–58, 1992.
24. F. Ranzato and F. Tapparo. Strong preservation as completeness in abstract interpretation. In D. Schmidt, editor, *Proc. of the 13th European Symposium on Programming (ESOP'04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 18–32. Springer-Verlag, 2004.
25. P. Ryan. Mathematical models of computer security – tutorial lectures. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 1–62. Springer-Verlag, 2001.
26. A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. of the International Symp. on Software Security (ISSS'03)*, Lecture Notes in Computer Science. Springer-Verlag, 2004.
27. A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE J. on selected ares in communications*, 21(1):5–19, 2003.
28. A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.
29. C. Skalka and S. Smith. Static enforcement of security with types. In *ICFP'00*, pages 254–267. ACM Press, New York, 2000.
30. D. Volpano. Safety versus secrecy. In *Proc. of the 6th Static Analysis Symp. (SAS'99)*, volume 1694 of *Lecture Notes in Computer Science*, pages 303–311. Springer-Verlag, 1999.
31. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.
32. S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 15–23. IEEE Computer Society Press, 2001.

Enforcing Resource Bounds via Static Verification of Dynamic Checks

Ajay Chander[1], David Espinosa[1], Nayeem Islam[1], Peter Lee[2], and George Necula[3]

[1] DoCoMo Labs USA, San Jose, CA
fax 408-573-1090
{chander, espinosa, islam}@docomolabs-usa.com
[2] Carnegie Mellon University, Pittsburgh, PA
Peter.Lee@cs.cmu.edu
[3] University of California, Berkeley, CA
necula@eecs.berkeley.edu

Abstract. We classify existing approaches to resource-bounds checking as static or dynamic. Dynamic checking performs checks *during* program execution, while static checking performs them *before* execution. Dynamic checking is easy to implement but incurs runtime cost. Static checking avoids runtime overhead but typically involves difficult, often incomplete program analyses. In particular, static checking is hard in the presence of dynamic data and complex program structure. We propose a new resource management paradigm that offers the best of both worlds. We present language constructs that let the code producer optimize dynamic checks by placing them either before each resource use, or at the start of the program, or anywhere in between. We show how the code consumer can then statically verify that the optimized dynamic checks enforce his resource bounds policy. We present a practical language that is designed to admit decidable yet efficient verification and prove that our procedure is sound and optimal. We describe our experience verifying a Java implementation of `tar` for resource safety. Finally, we outline how our method can improve the checking of other dynamic properties.

1 Introduction

Users are downloading code to run on their devices—computers, PDAs, cell phones, etc.—with increasing frequency. Examples of downloaded code include software updates, applications, games, active web pages, proxies for new protocols, codecs for new formats, and front-ends for distributed applications. At the same time, viruses, worms, and other malicious agents have also become common, resulting in attacks that include data corruption, privacy violation, and denial of service based on overuse of system resources. The latter problem is particularly relevant for small devices such as PDAs and cell phones. The state of the practice in mobile code execution includes powerful techniques that prevent data corruption (e.g., bytecode verification), but the enforcement of resource usage bounds is comparatively less developed. In this paper, we provide an efficient and flexible approach to limiting the resource usage of untrusted code. By *flexible*, we mean that our method applies to all sequential computer programs, including

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 311–325, 2005.

those where resource usage is not known until runtime. By *efficient*, we mean that it performs significantly fewer runtime checks while enforcing resource bounds than previous methods.

We address the scenario in which a *code consumer* runs an *untrusted* program created by a *code producer*, who possibly is untrusted. This program communicates with the code consumer's computer via a runtime library that provides functions to access resources. We consider both physical resources such as CPU, memory, disk, and network, as well as virtual resources such as files, database connections, and processes. Our goal is to limit resources according to the code consumer's *security policy*. This policy specifies the resources that each program can use, along with the corresponding usage bounds.

Our technique enforces resource usage bounds with a combination of static and dynamic checks. More precisely, we verify statically that a program's dynamic checks are sufficient to enforce the consumer's safety policy. In order to support such hybrid checking, we separate the `acquire` function, which *acquires* a resource, from the various functions that *consume* the resource. For notational simplicity, we use a single, specific `consume` function to represent abstractly any library function that consumes resources.

In current libraries, `acquire` and `consume` are performed together when a resource is used. It is easy to automatically replace these calls by pairs of separate calls to `acquire` and `consume`. It is also easy to verify statically that the result of this transformation never uses more resources than have been acquired.

The advantage of this separation is that the programmer, or appropriate optimization tools, can combine multiple `acquires` into one and can hoist `acquire` out of a loop whose body consumes resources. In this paper, we describe a static analysis that verifies that an arbitrary placement of `acquires` is sufficient. The analysis is decidable and efficient, and our experiments show that it can validate even aggressive optimizations. Moving `acquire` out of a loop can yield an arbitrary improvement in in the number of dynamic checks. This improvement results in significant performance gains if the `acquire` operation consults a complex or remote resource manager. Moving checks earlier can also guarantee that no resource errors occur in critical code fragments such as atomic transactions.

We begin this paper by introducing an imperative language with resource-aware constructs in Section 2, and illustrate the benefits of our method over purely static or dynamic approaches using a few key examples. In Section 3, we present an operational semantics for our language, and provide a precise characterization of resource-use safety. Section 4 describes the two components of the verifier: the safety condition generator (SCG) (Section 4.1) and the prover (Section 4.4), and presents soundness and optimality results for our SCG. We describe our experience with the `tar` program in Section 5. Section 6 positions this paper with respect to relevant work in a few areas. We mention ongoing efforts and future work opportunities in Section 7, and conclude in Section 8.

2 Concept

For resource-usage safety, we must ensure that each resource consuming operation, denoted by `consume`, has adequate resources available, as specified by a system security policy. Abstractly, we can refer to this policy as `quota`, and so the sum of all of the

consumes must be guaranteed never to exceed quota; we state this goal informally as consume $\leq$ quota. In the following, we motivate our approach to the resource-usage problem with a few examples, and introduce our method over a simple iterative language.

2.1 Examples

Figure 1 shows four programs that use resources. Program *Dynamic* uses an amount of resources that depends entirely on its runtime input. Program *Static* uses a fixed amount of resources. Program *Mixed1* uses a fixed amount of resources, but this amount is dynamic. Program *Mixed2* uses a fixed amount of resources each time through its inner loop, but it executes this loop a dynamic number of times.

A standard dynamic checker performs one check for each consume. It executes all four programs safely but adds unnecessary overhead to the static and mixed programs. A typical static analyzer adds no overhead to the static program but cannot execute the other three safely.

We present a method that has the advantages of both static and dynamic checkers. Like the dynamic checker, it safely executes all four programs. Like the static checker, it uses the static information available in each program to run more efficiently.

```
Program Dynamic
  while read() ≠ 0
    consume 1
```

```
Program Static
  i := 0
  while i < 10000
    consume 1
    i := i + 1
```

```
Program Mixed1
  N := read()
  i := 0
  while i < N
    consume 1
    i := i + 1
```

```
Program Mixed2
  while read() ≠ 0
    i := 0
    while i < 100
      consume 1
      i := i + 1
```

Fig. 1. Example programs

2.2 Language

In order to describe the static checking procedure, we use a simple imperative programming language that computes with integer values. Without loss of generality, we assume that there is one resource of interest whose amount is measured in some arbitrary unit. We introduce the command consume e to model any operation that uses e units of the resource, where e is an expression in the language. We introduce the command acquire e to reserve e resource units from the operating system. This command may fail, but if it succeeds, we know that e resource units have been reserved for the running program. The acquire operation is an example of a dynamic *reservation* instruction, perhaps realized with a library function, and occurs only in programs created by the code producer. In contrast, the consume operation acts as a no-op at execution time, and is used only in the static verification process.

$$
\begin{array}{lll}
e & ::= x \mid n \mid e_1 + e_2 \mid n * e \mid \texttt{cond}(b, e_1, e_2) & \text{Integer expressions} \\
b & ::= \texttt{true} \mid e_1 \geq e_2 \mid e_1 = e_2 & \text{Boolean expressions} \\
c & ::= \texttt{skip} \mid x := e \mid c_1; c_2 \mid & \\
 & \quad \texttt{consume}\ e \mid \texttt{acquire}\ e & \\
 & \quad \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid & \\
 & \quad \texttt{while}\ b\ \texttt{do}\ c\ \texttt{inv}\ (A, e) & \text{Commands} \\
P & ::= b \mid P_1 \wedge P_2 \mid A \Rightarrow P \mid & \\
 & \quad \forall x.P \mid \texttt{cond}(b, P_1, P_2) & \text{Predicates} \\
A & ::= b \mid A_1 \wedge A_2 & \text{Annotations}
\end{array}
$$

Fig. 2. Simple imperative language definition

```
Program Dynamic                 Program Static
  while read() ≠ 0                acquire 10000
    acquire 1                     i := 0
    consume 1                     while i < 10000
  inv (true, 0)                     consume 1
                                    i := i + 1
                                  inv (i ≤ 10000, 10000 − i)

Program Mixed1                  Program Mixed2
  N := read()                     while read() ≠ 0
  acquire N                         acquire 100
  i := 0                            i := 0
  while i < N                       while i < 100
    consume 1                         consume 1
    i := i + 1                        i := i + 1
  inv (i ≤ N, N − i)              inv (i ≤ 100, 100 − i)
```

Fig. 3. Example programs with annotations

Figure 2 shows the syntax of the full language. We assume that the variables x take only integer values. The expression $\texttt{cond}(b, e_1, e_2)$ has value e_1 if the boolean expression b has value `true` and has value e_2 otherwise. Similarly, the command $\texttt{cond}(b, P_1, P_2)$ is equivalent to the command P_1 if b has value `true`, and command P_2 otherwise. The propositional connectives $\wedge, \forall, \Rightarrow$ have their usual meaning.

The argument e to `acquire` and `consume` must be non-negative. The safety condition generator of Section 4.1 statically guarantees this condition.

Note also that we annotate the looping command with an invariant (A, e). During static checking, we verify that the predicate A holds and there are at least e resource units available before the looping command is executed. To simplify the task of the static checker, and to allow for a prover that is complete over safety conditions generated from programs in this language, we restrict the invariants to a conjunction of boolean equalities and comparisons between integer expressions and we similarly restrict the left side of implications in predicates.

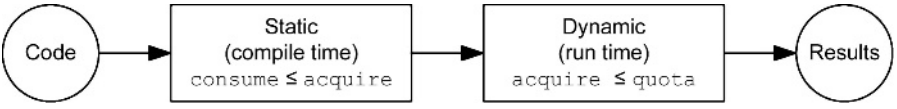


Fig. 4. Partitioning code safety into static and dynamic components

2.3 Annotated Examples

The programmer's (or automated tool's) job is to insert enough `acquire` operations to make the program safe. It is always possible to insert an `acquire` before each `consume`, so that each `consume` performs a runtime check, bringing us to the pure dynamic checking safety paradigm. The question is whether the programmer or automated tool can insert fewer `acquire` operations and thereby reduce the cost of dynamic checking.

Figure 3 shows the same four programs with `acquire` operations added. Note that all four programs execute safely. *Dynamic* performs exactly the same checks that it would in a dynamic system, acquiring each resource just before using it. *Static* performs exactly one check at the very beginning of execution. *Mixed1* and *Mixed2* perform far fewer checks than they would in a dynamic system, reserving all resources either at the beginning or each time through the outer loop; for example, *Mixed2* performs two orders of magnitude fewer checks.

Notice that the new language abstractions provide us with a midpoint in the original resource-usage condition `consume` $\leq$ `quota`. That is, we check statically that `consume` $\leq$ `acquire`, and we check dynamically that `acquire` $\leq$ `quota`. Figure 4 illustrates this concept. Static checking lets us hoist and combine `acquires`, so that we can use dynamically fewer of them and thus reduce the cost of checking.

3 Semantics of Annotated Programs

In this section, we formalize the meaning of expressions and commands, and make explicit the precise ways in which execution can fail, following well-known approaches to operational specifications of programming language constructs [1].

The execution state is a pair $\langle \sigma, n \rangle$ of an environment σ that maps variable names to integer values and a natural number n that represents the amount of available resources. We write $[\![e]\!]\sigma$ for the value of the integer expression e in the environment σ and $[\![b]\!]\sigma$ for the value of the boolean expression b in the environment σ. For example,

$$[\![\texttt{cond}(b, e_1, e2)]\!]\sigma = \begin{cases} [\![e_1]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{true} \\ [\![e_2]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \end{cases}$$

The other cases of the definition are straightforward. We use the notation $\sigma[x := n]$ to denote the environment that is identical to σ except that x is set to n.

3.1 Operational Semantics

We define the operational semantics of our language in terms of the judgment $\langle c, \sigma, n \rangle \Downarrow R$, which means that the evaluation of command c starting in state $\langle \sigma, n \rangle$ terminates

$$\frac{\sigma \not\models P}{\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{inv}\ (P,e),\sigma,n\rangle \Downarrow \texttt{InvFailure}}\ \text{WHILEINVFAILURE}$$

$$\frac{[\![b]\!]\sigma = \texttt{false} \quad \sigma \models P \quad n \geq [\![e]\!]\sigma}{\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{inv}\ (P,e),\sigma,n\rangle \Downarrow \langle \sigma,n\rangle}\ \text{WHILEF}$$

$$\frac{\begin{array}{l}\sigma \models P\ \ n \geq [\![e]\!]\sigma\ \ \langle c,\sigma,n\rangle \Downarrow \langle \sigma',n'\rangle \quad [\![b]\!]\sigma = \texttt{true}\ \ \sigma' \models P\ \ n' \geq [\![e]\!]\sigma' \\ \langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{inv}\ (P,e),\sigma',n'\rangle \Downarrow R\end{array}}{\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{inv}\ (P,e),\sigma,n\rangle \Downarrow R}\ \text{WHILET}$$

Fig. 5. Operational semantics for while loops

$$\frac{n \geq [\![e]\!]\sigma}{\langle \texttt{consume}e,\sigma,n\rangle \Downarrow \langle \sigma, n - [\![e]\!]\sigma\rangle}\ \text{C-OK} \qquad \frac{n < [\![e]\!]\sigma}{\langle \texttt{consume}e,\sigma,n\rangle \Downarrow \texttt{QuotaExceeded}}\ \text{C-FAIL}$$

$$\frac{}{\langle \texttt{acquire}e,\sigma,n\rangle \Downarrow \langle \sigma, n + [\![e]\!]\sigma\rangle}\ \text{A-OK} \qquad \frac{}{\langle \texttt{acquire}e,\sigma,n\rangle \Downarrow \texttt{AcquireFailed}}\ \text{A-FAIL}$$

Fig. 6. Operational semantics for reservations

with result R. If there does not exist an R such that $\langle c,\sigma,n\rangle \Downarrow R$, we write $\langle c,\sigma,n\rangle \Uparrow$ (pronounced "diverges").

The result R can be one of the following types of values. If the command terminates normally, then R is a new state $\langle \sigma', n'\rangle$. If an acquire fails, then R is the error `AcquireFailed`. If the program uses more resources than it has acquired, then R is the error `QuotaExceeded`. If the program does not satisfy an invariant annotation, then R is the error `InvFailure`. Thus, from an initial state, a command either diverges, terminates normally, or terminates with one of three errors.

Figure 5 shows the operational semantics for while loops; the operational semantics for the other standard constructs is straightforward.

Figure 6 shows the rules for evaluating resource-specific commands, which modify the amount of resources in the current state. Notice that only `acquire` replenishes this state, so that if the program starts with no resources, it must `acquire` all the resources that it uses. If enough resources are available, `consume` terminates normally, consuming resources. If not enough resources are available, it yields a `QuotaExceeded` error. The `acquire` command either increases the amount of available resources or yields an `AcquireFailed` error. In this formalization, the `acquire` command is nondeterministic. In practice, its behavior is determined by the operating system, which we do not model here. Alternatively, we could add an explicit dynamic pool to model the resources available to `acquire`.

4 Verifier

The verifier has two parts, the safety condition generator (SCG), which computes a program's safety condition (SC), and the prover, which actually proves the SC. We define safety, state a soundness theorem, which says that the SC guarantees safety, and state an optimality theorem, which says that the SC captures *all* programs that are safe.

$$
\begin{array}{ll}
\texttt{scg}(\texttt{skip})(P,e) & = (P,e) \\
\texttt{scg}(c_1;c_2)(P,e) & = \texttt{scg}(c_1)(\texttt{scg}(c_2)(P,e)) \\
\texttt{scg}(x := e')(P,e) & = ([e'/x]P, [e'/x]e) \\
\texttt{scg}(\texttt{consume}e')(P,e) & = (P \wedge e' \geq 0, e' + \texttt{cond}(e \geq 0, e, 0) \\
\texttt{scg}(\texttt{acquire}\ e')(P,e) & = (P \wedge e' \geq 0, e - e') \\
\texttt{scg}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)(P,e) & = (\texttt{cond}(b, P_1, P_2), \texttt{cond}(b, e_1, e_2)) \\
 & \quad \text{where } (P_1, e_1) = \texttt{scg}(c_1)(P,e) \\
 & \quad \text{and } \ (P_2, e_2) = \texttt{scg}(c_2)(P,e) \\
\texttt{scg}(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{inv}\ (A_I, e_I))(P,e) & = (A_I \wedge \forall \boldsymbol{x}. A_I \Rightarrow \texttt{cond}(b, Q', Q), e_I) \\
 & \quad \text{where } (P', e') = \texttt{scg}(c)(A_I, e_I) \\
 & \quad \text{and } \ Q' = P' \wedge e_I \geq e' \\
 & \quad \text{and } \ Q = P \wedge e_I \geq e \\
 & \quad \text{and } \ \boldsymbol{x} \text{ are the variables modified in } c
\end{array}
$$

Fig. 7. Definition of `scg`

4.1 Safety Condition Generator

Our verifier uses a variant of Dijkstra's weakest precondition calculus [2]. We work with "generalized predicates" (P, e), meaning that P holds and there are at least e resource units available. Figure 7 shows the definition of the safety condition generator `scg`. We define `scg` by recursion on the syntax of commands. Our definition matches the standard `scg` definition for all commands that do not manipulate resources explicitly. For the commands that manipulate resources, we extracted the definition from the soundness proof. The `scg` definition also (1) checks the invariant that there are a non-negative amount of resources available and (2) checks that the arguments to `acquire` and `consume` are non-negative.

Although our language uses structured control (`while` loops), we can also define `scg` for unstructured control (`gotos`), by associating an invariant with each label, or at least those at the heads of loops, as determined by a standard dominator-based control flow analysis.

4.2 Soundness

We write $\sigma \models P$ to indicate that predicate P holds in state σ. Recall that σ supplies values for the variables in P, so we define $\sigma \models P$ as usual by induction over the propositional connectives.

Definition 1. $\langle \sigma, n \rangle \models (P, e)$ *iff* $n \geq 0$, $\sigma \models P$, *and* $\sigma \models n \geq e$.

That is, n is non-negative, P holds in σ, and at least e resources are available in σ.

Definition 2. *A tuple* $(c, \langle \sigma, n \rangle, (P, e))$ *is* safe *iff one of the following holds:*

1. $\langle c, \sigma, n \rangle \Uparrow$, *or*
2. $\langle c, \sigma, n \rangle \Downarrow \texttt{AcquireFailed}$, *or*
3. $\langle c, \sigma, n \rangle \Downarrow \langle \sigma', n' \rangle$ *where* $\langle \sigma', n' \rangle \models (P, e)$.

That is, the `InvFailure` and `QuotaExceeded` errors do not occur, and (P, e) holds if execution terminates normally.

Definition 3. *A tuple* $(c, (P_0, e_0), (P, e))$ *is* safe *iff* $(c, \langle \sigma, n \rangle, (P, e))$ *is safe for all states* $\langle \sigma, n \rangle$ *such that* $\langle \sigma, n \rangle \models (P_0, e_0)$.

That is, (P_0, e_0) guarantees safety. We can now state the soundness theorem:

Theorem 1. *For all* $c, P, e,$ $(c, \texttt{scg}(c)(P, e), (P, e))$ *is safe.*

Proof: By structural induction on the derivation $\langle c, \sigma, n \rangle \Downarrow R$.

To check whether a given command is safe to execute, we check whether $\langle \sigma_0, 0 \rangle \models \texttt{scg}(c)(\texttt{true}, 0)$. That is, we compute the command's safety condition and check whether it holds in the initial execution state σ_0. If it holds, then c cannot produce the `InvFailure` or `QuotaExceeded` errors. Thus, we do not check for them dynamically in actual practice, and we do not maintain the static resource pool n. Note that we *do* still check for dynamic policy violations, which raise the `AcquireFailed` error.

4.3 Optimality

The soundness theorem shows that our `scg` prevents c from raising the `InvFailure` or `QuotaExceeded` errors. The `scg` is also optimal, meaning that it generates the weakest such condition, in the sense of:

Definition 4. $(P_0, e_0) \Rrightarrow (P_1, e_1)$ *iff* $P_0 \Rightarrow (P_1 \wedge e_0 \geq e_1)$.

That is, whenever (P_0, e_0) holds, so does (P_1, e_1). The optimality theorem states:

Theorem 2. *For all* $c, P_0, e_0, P, e,$ *if* $(c, (P_0, e_0), (P, e))$ *is safe, then* $(P_0, e_0) \Rrightarrow \texttt{scg}(c)(P, e)$.

Proof: By structural induction on the command c.

That is, whenever (P_0, e_0) guarantees safety, (P_0, e_0) implies $\texttt{scg}(c)\ (P, e)$. Thus, $\texttt{scg}(c)\ (P, e)$ is the weakest such condition.

4.4 Prover

In this section, we show how to prove the safety conditions. We observe that the grammar for predicates restricts the left side of implications to annotations, not full predicates. Annotations are conjunctions of boolean expressions that are equalities or comparisons between integer expressions.

We also observe that the our definition of `scg` respects this restriction. In particular, all formulas on the left side of an implication arise from loop invariants and pre and post conditions.

Thus, we use a simple theorem prover $\texttt{prove} : a \times p \rightarrow \mathit{Bool}$ where $\texttt{prove}(A, P)$ holds if and only if $A \Rightarrow P$ is valid. Valid means that the formula is true for all values of the global variables and fresh constants introduced by the rule for universal quantification. A predicate P is valid if and only if the $\texttt{prove}(\texttt{true}, P)$ is true. Figure 8 shows the definition of `prove`.

To prove $A \Rightarrow P$, `prove` recursively decomposes P until it reaches a boolean expression b. It then uses a satisfiability procedure `sat` to check whether $A \Rightarrow b$ is valid.

$$
\begin{array}{ll}
\texttt{prove}(A, b) & = \neg\texttt{sat}(A \wedge \neg b) \\
\texttt{prove}(A, P_1 \wedge P_2) & = \texttt{prove}(A, P_1) \wedge \texttt{prove}(A, P_2) \\
\texttt{prove}(A, A_1 \Rightarrow P) & = \texttt{prove}(A \wedge A_1, P) \\
\texttt{prove}(A, \forall x.P) & = \texttt{prove}(A, [a/x]P) \ \ (a \text{ is fresh}) \\
\texttt{prove}(A, \texttt{cond}(b, P_1, P_2)) & = \texttt{prove}(A \wedge b, P_1) \wedge \texttt{prove}(A \wedge \neg b, P_2)
\end{array}
$$

Fig. 8. Definition of `prove`

As usual, $A \Rightarrow b$ is valid if and only if its negation $A \wedge \neg b$ is unsatisfiable. Since the form of A is restricted, we only call `sat` on a conjunction of (possibly negated) boolean expressions. Since `prove` decomposes P using invertible rules, it is sound and complete if and only if `sat` is sound and complete.

There are two notable satisfiability procedures that handle the linear inequalities that `scg` generates. One is due to Nelson and Oppen [3] and implemented by the Simplify prover [4] used in ESC/Java [5]. The other is due to Shostak [6] and implemented in PVS [7]. In our experiments, we used ESC/Java and Simplify to generate and prove safety conditions from Java code. For our class of conditions, Simplify is sound and complete.

Although we can probably trust our simple recursive prover, we may not want to trust the more complex satisfiability procedure at its core. To address this problem, we can use proof-carrying code [8] and require the program producer to send a safety proof to the program consumer. If the satisfiability procedure generates verifiable proofs, then the producer can create a safety proof by running the `prove` procedure and collecting all the satisfiability proofs. The program consumer can check the proof by running the `prove` procedure, just as the producer did, and checking each of the satisfiability proofs. We may also choose to use PCC if if we enrich the language of invariants and replace our simple prover with a more complex first-order prover.

4.5 Annotator

As it stands, our approach requires the programmer manually to insert `acquire`s, write loop invariants, and add function pre and post conditions. We are currently working on a tool that automatically and correctly adds these assertions, similar in spirit to Houdini [9]. Although optimal annotation is undecidable, the tool can "fall back" to inserting an `acquire` before each `consume`. This annotation scheme is verifiable using the trivial loop invariant `true`, and it removes the need for hand annotation when the programmer does not care about efficiency. Beyond this "base line" performance, we plan to include a knowledge base of common loop idioms and their optimal annotations.

One advantage of manual annotation is that the programmer can decide how early to acquire resources. It is less costly to acquire all resources at once, but it is also "anti-social" to hold unused resources, preventing other concurrently running programs from using them. The programmer can also decide whether to acquire exactly the right amount of resources, which may be difficult to determine, or whether to over-estimate.

4.6 Renewable Resources

We can easily extend our framework to handle renewable resources, such as memory and file handles, by allowing `acquire` and `consume` to take negative arguments. In essence, `acquire` and `consume` manage two pools, a *static pool* and a *dynamic pool*. With a positive argument, `acquire` moves resources from the dynamic pool to the static pool. With a positive argument, `consume` moves resources out of the static pool. With negative arguments, `acquire` and `consume` transfer resources in the opposite direction. The `consume` operation is part of the TCB in that it represents (or annotates) trusted library functions such as `malloc` (with positive argument) and `free` (with negative argument). The `acquire` operation is untrusted, and downloaded code is free to call it to obtain resources from the run-time system, or to release them back to the run-time system.

5 `Tar` Example

In this section, we describe our experience with a version of `tar` written in Java. We wanted to see how hard it would be to annotate a "real" program, whether we could report policy violations earlier, and whether we could reduce the cost of dynamic checks. We chose a security policy that limits the number of bytes that `tar` reads and writes to the file system.

We began with a Java `tar` program from ICE Engineering [10] but revised it to simplify the annotation process. Although `tar` contains 1700 lines of code, we only needed to examine the 577 lines relevant to I/O.

We prototyped our ideas using ESC/Java [5], which checks pre and post conditions for Java code using the Simplify theorem prover [4]. Using the definitions of `acquire` and `consume` shown in Figure 9, ESC/Java generates essentially the same verification condition as the function shown in Figure 7. Although ESC/Java has been excellent for prototyping our ideas, it is not suitable for verifying code safety. First, it is unsound, because it does not throughly check loop invariants and side-effect assertions (`modifies`). Thus, it cannot form the basis for a secure system. Second, it does not generate certificates for later verification. Third, it is too large to run on mobile devices. For these reasons, we are developing a lightweight implementation based on a certificate-generating prover [11].

The implementation of our ideas in ESC/Java is straightforward. We maintain two pools, a static pool and a dynamic pool. We represent the static pool using a *ghost variable* that exists only at verification time. At the start of execution, the user's security policy fills the dynamic pool with the program's resource quota. The `acquire` operation transfers resources from the dynamic pool to the static pool. The `consume` operation removes resources from the static pool. The invariants ensure that the pools never drop below empty. Note that ESC/Java verifies each method's implementation against its specification using only the specifications, not the implementations, of the methods that it calls.

```
1  private static long dynamicPoolRead = 0;
2  //@ ghost public static long staticPoolRead = 0;
3  //@ invariant staticPoolRead >= 0;
4  //@ invariant dynamicPoolRead >= 0;
5
6  //@ requires n >= 0;
7  //@ ensures staticPoolRead == \old(staticPoolRead) + n;
8  //@ modifies dynamicPoolRead, staticPoolRead;
9  public static void acquireRead (long n) {
10   if (dynamicPoolRead >= n) {
11     dynamicPoolRead -= n;
12     //@ set staticPoolRead = staticPoolRead + n;
13   } else {
14     System.out.println ("Read quota exceeded!\n");
15     System.exit (1);
16   }
17 }
18
19 //@ requires n >= 0 && staticPoolRead >= n;
20 //@ ensures staticPoolRead == \old(staticPoolRead) - n;
21 //@ modifies staticPoolRead;
22 public static void consumeRead (long n) {
23   //@ set staticPoolRead = staticPoolRead - n;
24 }
```

Fig. 9. Implementation of `acquire` and `consume` in ESC/Java

```
1  long size = file.length ();
2  long q = size / recordSize;
3  long r = size % recordSize;
4  long size2 = q * recordSize;
5  long size3 = size2 + (r > 0) ? recordSize : 0;
6
7  Resources.acquireWrite (size3 + recordSize);
8  Resources.acquireRead (size);
9  out.writeHeaderRecord (entry);
10
11 for (int i = 0; i < q; ++i) {
12   in.read (buffer, 0, recordSize);
13   out.writeRecord (buffer);
14 }
15
16 if (r > 0) {
17   Arrays.fill (buffer, (byte) 0);
18   in.read (buffer, 0, r);
19   out.writeRecord (buffer);
20 }
```

Fig. 10. Java `tar` code excerpt

The naive `tar` implementation requires two dynamic checks for each 512-byte block, one for `read` and one for `write`. Using reservations, our implementation perform two checks per *file* rather than two checks per *block*. Figure 10 shows the code to write a file to the archive. We replaced the usual "while not EOF" loop with a `for` loop that counts a definite number of blocks. This structure ensures that `tar` does not exceed its quota even if a concurrent process lengthens the file.

Ideally, we would like to perform only two checks to create the entire archive. We haven't tried this experiment yet, but the code would need to prescan the directories to build a table of file sizes. The prover would need to connect the loop that sums the file sizes to the loop that reads the files.

We annotated each I/O method by computing its resource use in terms of the resource use of its subroutines. If a method's use was dynamic or difficult to state in closed form, we added a dynamic check to stop its upward propagation ("the buck stops here"). Although we experimented with annotations that overestimate resource use, we found that precise annotations were simple enough. In total, `tar` required 33 lines of annotation.

We tested `tar` on a directory containing 13.4 mb in 1169 files, for an average file size of 11.7 kb. The unannotated program performed 57210 I/O operations on 512-byte blocks. Since each operation requires a dynamic check, it also performed 57210 dynamic checks. The annotated program also performed 57210 I/O operations. However, since it performed one dynamic check per file rather than per block, it only performed 2389 dynamic checks. That is, it performed almost 24 times fewer dynamic checks. Of course, this ratio is the average file size divided by 512.

Because block I/O operations are much more expensive than dynamic checks, we did not obtain a corresponding decrease in overall run time. However, our technique also applies to operations where the check is expensive relative to the operation itself, such as instruction counting and memory reference counting.

6 Related Work

Our work combines ideas from several areas: Dijkstra's weakest precondition computation [2], Necula and Lee's proof-carrying code [8], partial evaluation's separation of static and dynamic binding times [12], and standard compiler optimizations such as hoisting and array bounds check elimination [13].

Since we *combine* static and dynamic checking, our work is only tangentially related to purely static approaches such as Crary and Weirich's resource bound certification [14] or purely dynamic approaches such as the Java security monitor [15]. The implementations based on bytecode rewriting [16, 17, 18, 19, 20, 21, 22] are also purely dynamic, since they add checks without performing significant static analysis.

Our approach is a non-trivial instance of Necula and Lee's safe kernel extension method [23]. They show that the OS designer can export an unsafe, low-level API if he provides a set of rules for its use, and a static analysis that checks whether clients follow these rules. By contrast, most designers wrap the low-level API in a safe but inefficient high-level API that clients can call without restriction. For array bounds checking, the

low-level API is the unguarded reference, while the high-level API guards the reference with a bounds check. The usage rule is that the index must be in bounds.

In our case, the low-level API is `acquire` and `consume`. The high-level API, which we intentionally avoid, immediately prefixes `consume` by `acquire`, so that each `consume` has enough resources. This high-level API provides pure dynamic checking. The usage rule is that we `acquire` some time before we `consume`, but not necessarily immediately before. We extricate this useful, low-level API from its high-level wrapper and provide a flexible but safe set of usage rules, which we show how to statically check efficiently. The end result is a novel combination of static and dynamic checking.

On the surface, our work seems similar to approaches that place dynamic checks according to static analysis, such as Wallach's SAFKASI system [24] and Gupta's elimination and hoisting of array bounds checks [13]. These systems limit the programmer to the safe, high-level API, but they inline and optimize calls to it according to the low-level API's usage rules and semantics. By contrast, like PCC, we separate verification from optimization, which is untrusted and can be performed by the programmer or by an automated tool. The programmer can also ignore the high-level API and call the low-level API directly.

Like us, Patel and Lepreau [25] describe a hybrid (mixed static and dynamic) approach to resource accounting. They use static analysis of execution time to reject some overly expensive active network router extensions. They use dynamic checks to monitor other, unspecified resources. At this level of detail, their static and dynamic checks are not tightly coupled. However, they also use static analysis to locate dynamic network polling operations. They bind their ideas closely to the complex active network setting and do not extract a simple, reuseable API or a proof system for reasoning about it.

Independently of us, Vanderwaart and Crary proposed the TALT-R system [26]. They place a *yield* at most every Y instructions. That is, *yield* is similar to `acquire`(Y). However, since *yield* does not itself debit a resource quota, it does not enable the fine-grained combination of static and dynamic checking.

7 Extensions and Future Work

Our approach can already handle (1) function pre and post conditions and (2) reuseable resources such as memory, but we do not have space to describe these extensions here.

We are currently engaged in future work in several different areas. First, due to the limitations of existing tools, we are developing an SCG and prover that can prove resource-use safety for Java bytecode and produce proof witnesses. This effort presents several engineering challenges, such as scaling our SCG to a larger language, tracking source level annotations in bytecode, and building an efficient proof checker that performs well on mobile devices. Second, we are designing a tool that automatically and correctly annotates bytecode with resource reservations. Third, we are applying our techniques to other security mechanisms such as stack inspection and access control. Fourth, we are investigating situations where the check is expensive relative to the operation itself, such as instruction counting and memory reference sandboxing.

8 Conclusion

We have demonstrated a novel API for resource bounds enforcement that combines the best of static and dynamic approaches, by providing the means to dynamically reserve resources within programs and statically verify that the reservations are sufficient. Our soundness theorem gives the code consumer total confidence that statically verified programs do not exceed the resource bounds specified in his safety policy. Our approach gives the code producer (programmer or automated tool) complete freedom to optimize the placement of dynamic checks. Thus, we provide a system for writing statically verifiable resource-safe programs that handles dynamic data and complex program structure.

By adapting ideas from weakest preconditions and proof-carrying code, we showed how the code consumer can statically verify that resource reservations enforce his resource bounds policy. We presented a practical language that was carefully designed to admit decidable yet efficient verification and proved soundness and optimality theorems. Finally, we described our experience in successfully annotating and verifying a Java version of `tar` for resource safety.

Furthermore, our approach generalizes to APIs other than resource checking. At present, code consumers hide these APIs in high-level wrappers that are safe but inefficient. Using our hybrid approach, code consumers can give code producers direct access to efficient, low-level APIs without sacrificing safety.

References

1. Mitchell, J.C.: Foundations for Programming Languages. MIT Press (1996)
2. Dijkstra, E.: A Discipline of Programming. Prentice-Hall (1976)
3. Nelson, G., Oppen, D.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems **1** (1979) 245–257
4. Detlefs, D., Nelson, G., Saxe, J.: Simplify: a theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories (2003)
5. Flanagan, C., Leino, R., Lilibridge, M., Nelson, G., Saxe, J., Stata, R.: Extended static checking for Java. In: Programming Language Design and Implementation, Berlin, Germany (2002)
6. Shostak, R.E.: Deciding combinations of theories. Journal of the ACM **31** (1984) 1–12
7. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In Kapur, D., ed.: 11th International Conference on Automated Deduction (CADE). Volume 607 of Lecture Notes in Artificial Intelligence., Saratoga, NY, Springer-Verlag (1992) 748–752
8. Necula, G.: Proof-carrying code. In: Principles of Programming Languages, Paris, France (1997)
9. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Formal Methods Europe (LNCS 2021), Berlin, Germany (2001)
10. Endres, T.: Java Tar 2.5. `http://www.trustice.com` (2003)
11. Necula, G.C., Rahul, S.P.: Oracle-based checking of untrusted software. In: Principles of Programming Languages, London, England (2001)
12. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall (1993)
13. Gupta, R.: Optimizing array bound checks using flow analysis. ACM Letters on Programming Languages and Systems **2** (1993) 135–150

14. Crary, K., Weirich, S.: Resource bound certification. In: Principles of Programming Languages, Boston, Massachusetts (2000)
15. Gong, L.: Inside Java 2 Platform Security. Addison-Wesley (1999)
16. Czajkowski, G., von Eicken, T.: JRes: a resource accounting interface for Java. In: Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, BC (1998)
17. Evans, D., Twyman, A.: Flexible policy-directed code safety. In: Security and Privacy, Oakland, California (1999)
18. Erlingsson, U., Schneider, F.: SASI enforcement of security policies: a retrospective. In: New Security Paradigms Workshop, Caledon, Canada (1999)
19. Colcombet, T., Fradet, P.: Enforcing trace properties by program transformation. In: Principles of Programming Languages, Boston, Massachusetts (2000)
20. Pandey, R., Hashii, B.: Providing fine-grained access control for Java programs via binary editing. Concurrency: Practice and Experience **12** (2000) 1405–1430
21. Chander, A., Mitchell, J., Shin, I.: Mobile code security by Java bytecode instrumentation. In: DARPA Information Survivability Confernce and Exposition. (2001)
22. Kim, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: a run-time assurance tool for Java programs. Electronic Notes in Theoretical Computer Science **55** (2001)
23. Necula, G., Lee, P.: Safe kernel extensions without run-time checking. In: Operating Systems Design and Implementation, Seattle, Washington (1996)
24. Wallach, D., Appel, A., Felten, E.: SAFKASI: a security mechanism for language-based systems. Transactions on Software Engineering **9** (2000) 341–378
25. Patel, P., Lepreau, J.: Hybrid resource control of active extensions. In: Open Architectures and Network Programming, San Francisco, California (2003)
26. Vanderwaart, J., Crary, K.: Foundational typed assembly language for grid computing. Technical Report CMU-CS-04-104, Carnegie-Mellon University (2004)

Asserting Bytecode Safety

Martin Wildmoser and Tobias Nipkow

Technische Universität München, Institut für Informatik
{wildmosm, nipkow}@in.tum.de

Abstract. We instantiate an Isabelle/HOL framework for proof carrying code to Jinja bytecode, a downsized variant of Java bytecode featuring objects, inheritance, method calls and exceptions. Bytecode annotated in a first order expression language can be certified not to produce arithmetic overflows. For this purpose we use a generic verification condition generator, which we have proven correct and relatively complete.

1 Proof Carrying Code

In mobile code applications, e.g. applets, grid computing, dynamic drivers, or ubiquitous computing, safety is a primary concern. Proof carrying code (PCC) aims at certifying that low level code adheres to some safety policy, such as type safety [6], bounded array accesses [13], or limited memory consumption [4]. When such properties are checked statically sandbox mechanisms and error recovery become obsolete. In classical PCC a verification condition generator (VCG) reduces annotated machine code to proof obligations that guarantee safety. Proofs, usually obtained automatically with a theorem prover, are then shipped to the code consumer, who checks them. The whole setup is sound if the VCG and the proof checker can be trusted. In Foundational Proof Carrying Code [3] the VCG is eliminated by proving safety directly on the machine semantics, typically assisted by a source level type system. Our approach is to formalize and verify PCC in a theorem prover. In [19] we present an Isabelle/HOL [15] framework for PCC. The essential part is a generic, executable and verified VCG. This turns out to be feasible as only a small part of a VCG needs to be trusted. Many parts can be outsourced in form of parameters that can be customized to the programming language, safety policy and safety logic at hand. In this paper we instantiate a PCC system for Jinja bytecode and a safety policy that prohibits arithmetic overflow. We verified that this instantiation meets all the requirements our framework demands for the VCG to be correct and relatively complete. Verifying programs at the bytecode level has clear advantages. First, one does not have to trust a compiler. Second, the source code, which is often kept secret, is not required. Third, many safety policies are influenced by the machine design. For example verifying sharp runtime bounds even requires going down to the processor level and considering pipeline and caching activities. In case of bytecode, we need a safety logic that can adequately model JVM states. Over the years various logics for object oriented programs have been proposed.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 326–341, 2005.

For instance [2] defines a Hoare Logic based on a combination of a type system and first order logic with equality. In [16] a shallow embedding of Isabelle/HOL is used to define a Hoare logic for Java. A very prominent annotation language for Java is JML [10] or the downsized version of it used in ESC Java [7]. However, all of the logics have been designed for Java, not its bytecode. This paper introduces a first order expression language with JVM specific constructs. This language is expressive enough for weakest preconditions of Jinja instructions and a safety policy against arithmetic overflow. Although this is undecidable, many programs produce proof obligations that are easy enough to be handled by Isabelle/HOL's decision procedures, such as Cooper's algorithm [14].

2 Jinja Bytecode

Jinja bytecode is a downsized version of Java bytecode. Although it only has 16 instructions, it covers most object oriented features: object creation, inheritance, dynamic method calls and exceptions.

datatype *instr* =

`Load` *nat*	load from register
| `Store` *nat*	store into register
| `Push` *val*	push a constant
| `New` *cname*	create object on heap
| `Getfield` *vname cname*	fetch field from object
| `Putfield` *vname cname*	set field in object
| `Checkcast` *cname*	check if object is of class *cname*
| `Invoke` *mname nat*	invoke instance method with *nat* parameters
| `Return`	return from method
| `Pop`	remove top element
| `IAdd`	integer addition
| `Goto` *int*	goto relative address
| `CmpEq`	equality comparison
| `IfFalse` *int*	branch if top of stack false
| `IfIntLeq` *int*	take integers a and b from stack, branch if $a \leq b$
| `Throw`	throw exception

Jinja programs are interpreted by the Jinja virtual machine, which closely models the Java VM. States consist of a flag indicating whether an exception is raised (if yes, a reference to the exception object), a heap and a method frame stack.

types *jvm-state* = *addr option* × *heap* × *frame list*

The heap is a partial map from addresses (natural numbers) to objects. We use the polymorphic type $'a\ option = None \mid Some\ 'a$ to model partiality in Isabelle/HOL, a logic of total functions. Using *the* one can extract the content,e.g. $the\ (Some\ a) = a$.

types *heap* = *addr* ⇒ *obj option*
obj = *cname* × *fields*

fields = (*vname* × *cname*) ⇒ *val option*
cname = *vname* = *mname* = *string*

Jinja has values for booleans, e.g. *Bool True*, integers, e.g. *Intg 5*, references, e.g. *Addr 3*, null pointers, e.g. *Null* or dummy elements, e.g. *Unit*. For some values, we use partially defined extractor functions.

datatype *val* = *Bool bool* | *Intg int* | *Addr addr* | *Null* | *Unit*
the-Intg (*Intg i*) = *i*, *the-Bool* (*Bool b*) = *b*, *the-Addr* (*Addr a*) = *a*

Each value has a type associated with it:

datatype *ty* = *Boolean* | *Integer* | *Class cname* | *NT* | *Void*

Whenever a method is called a new frame is allocated on the method frame stack. This frame contains registers, an operand stack and the program counter. In the registers the Jinja VM stores the **this** reference, the method's arguments and its local variables. The operand stack is used to evaluate expressions. Both are modelled as lists, e.g. $l = [Null, Unit]$, for which Isabelle/HOL provides many operators. For example, there is concatenation, e.g. $l@[Intg\ 2] = [Null, Unit, Intg\ 2]$, indexed lookup $l\ !\ 1 = Unit$, head,i.e. $hd\ l = Null$ and tail, $tl\ l = [Unit]$.

types *frame* = *opstack* × *registers* × *pos*
opstack = *val list*
registers = *val list*
pos = *cname* × *mname* × *nat*

Instructions are identified with positions. For example (C,M,pc) points to instruction number pc in method M of class C. Each method is a tuple of the form (mxs,mxr,is,et), where *mxs* indicates the maximum operand stack height, *mxr* the number of used registers, *is* the method body and *et* the exception table.

types *jvm-method* = *nat* × *nat* × *instr list* × *ex-table*

The exception table is a list of tuples $(f,\ t,\ C,\ h,\ d)$:

types *ex-table* = (*nat* × *nat* × *cname* × *nat* × *nat*) *list*

Whenever an intruction within the *try* block bounded by $[f,\ t)$ throws an exception of type C the handler starting at h is executed. The parameter d, which is always *0* in our case, specifies the size of the stack the handler expects. This is used in [9] to handle exceptions within expression evaluation, but is not required for real Java programs. Jinja programs are lists of class declarations. Each class declaration (C,S,fs,ms) consists of the name of the class, the name of its direct superclass, a list of field declarations, which are pairs of field names and types, and a list of method declarations. Method declarations consist of the method's name, its argument types, its result type and its body.

types *jvm-prog* = (*cname* × *cname* × *fdecl list* × *mdecl list*) *list*
fdecl = *vname* × *ty*
mdecl = *mname* × *ty list* × *ty* × *jvm-method*

Our PCC system requires programs Π with annotations. These are added by finite maps from positions to logical expressions. Finite maps are lists of pairs,

e.g. $fm = [(1,1),(3,5),(3,6)]$, and have operations for lookup, e.g. $fm{\downarrow}0 = None$ or $fm{\downarrow}3 = Some\ 5$, domain, e.g. $dom\ fm = [1,3,3]$ and range, e.g. $ran\ fm = [1,5,5]$. Note that a pair (x,y) is overwritten by a pair (x,y') to the left of it.

types *jbc-prog* = *jvm-prog* × (*pos* ▷ *expr*)

To specify and verify safety properties we need to reason about Jinja VM states. A central issue of this paper is to suggest a formula language for this purpose. For some constructs of this language we require an extended version of the Jinja VM, one which memorizes additional information in a so called environment e, in order to define their semantics. In addition our PCC framework requires positions to be given as the first component of a state.

types *jbc-state* = *pos* × *jvm-state* × *env*

The environment e contains a virtual stack of call states *cs* e and a binding *lv* e for so called logical variables.

record *env* =*cs*:: *heap list*
lv:: *var* ⇒ *val*

Whenever a new frame is allocated on the method call stack, we record the current heap and register values in a call stack, which acts like a history variable in Hoare Logics. Whenever a frame is popped, we also pop an entry from the call stack. The bytecode semantics consists of two rules: One specifies normal, the other one exceptional execution.

nrml:
⟦ *P* = *fst Π*; *p* = (*C*,*M*,*pc*); *i* = *instrs-of P C M* ! *pc*;
σ = (*None*,*h*,(*stk*,*loc*,*p*)#*frs*); *check P* σ;
exec-instr i P h stk loc C M pc frs = (*None*,*h'*,*fr'*#*frs'*);
σ' = (*None*,*h'*,*fr'* # *frs'*); *p'* = *snd* (*snd fr'*);
e' = *e*(|*cs*:=*if* ∃ *M n*. *i* = *Invoke M n then h*#(*cs e*)
else if i = *Return then tl* (*cs e*) *else cs e*|)
⟧ ⟹ ((*p*,σ,*e*),(*p'*,σ',*e'*)) ∈ *effS Π*

expt:
⟦ *P* = *fst Π*; *p* = (*C*,*M*,*pc*); *i* = *instrs-of P C M* ! *pc*;
σ = (*None*,*h*,(*stk*,*loc*,*p*)#*frs*); *check P* σ;
exec-instr i P h stk loc C M pc frs = (*Some xa*,-,-);
find-handler P xa h ((*stk*,*loc*,*p*)#*frs*) = σ';
σ' = (*None*,*h*,([*Addr xa*],*loc'*,*p'*)#*frs'*);
e' = *e*(|*cs*:=*drop* (*length frs* − *length frs'*) (*cs e*)|)
⟧ ⟹ ((*p*,σ,*e*),(*p'*,σ',*e'*)) ∈ *effS Π*

In both rules we use *instrs-of* to retrieve the instruction list of the current method. The actual execution for single instructions is delegated to *exec-instr*, whose full definition can be found in [9]. Here we only give one example:

exec-instr IAdd P h stk loc C_0 M_0 *pc frs* =
(*let* i_2 = *the-Intg* (*hd stk*); i_1 = *the-Intg* (*hd* (*tl stk*))
in (*None*, *h*, (*Intg* (i_1+i_2)#(*tl* (*tl stk*)), *loc*, C_0, M_0, *pc*+1)#*frs*))

The function *exec-instr* returns triples, whose first component indicates whether an exception occurs. If yes (second rule), we use the function *find-handler*) to do exception handling similar to the Java VM: it looks up the exception table in the current method, and sets the program counter to the first handler that protects *pc* and that matches the exception class. If there is no such handler, the topmost call frame is popped, and the search continues recursively in the invoking frame. If no exception handler is found, the exception flag remains set and the machine halts. If this procedure does find an exception handler (f, t, C, h, $0{::}'e$) it sets the *pc* to h and empties the operand stack except for reference to the exception object. Additional safety checks in the semantics, i.e. *check* P σ, ensure that arguments of proper type and number are used. This simplifies the verification of the soundness and completness requirements our PCC framework demands. As proven in [9] the bytecode verifier only accepts programs for which these checks hold. Hence, it is sound to work with this defensive version of the Jinja VM. We require Jinja Bytecode programs to have a main method. For simplicity we assume that this method is named *main*, has no arguments, and belongs to a class called *Start*. This means we start a program at position (*Start*,*main*,*0*). The initial operand stack is empty and the registers are initialized with arbitrary values. The initial heap (*start-heap* (*fst* Π)) contains three entries for system exceptions *NullPointer*, *ClassCast* and *OutOfMemory*. The initial environment contains an empty call stack. The binding of logical variables is unrestricted.

initS $\Pi \equiv \{\ (p,\sigma,e).\ p = (Start,main,0) \wedge cs\ e = []$
$\wedge\ \sigma = (None,start\text{-}heap\ (fst\ \Pi),[([],Null\ \#\ replicate\ mxr\ arbitrary,p)])\ \}$

3 Assertion Logic

Many aspects of Java are identical in the bytecode, but an important concept of the Java VM cannot be found at the source level: The method frame stack. This stack is used to store local data of methods in evaluation. Almost all bytecode instructions affect it. If we want to simulate these effects at a syntactic level, we need a language that can describe this stack. Fig. 1 shows the assertion language we use for this purpose. It is a variant of first order arithmetic with special constructs for adequate modeling of JVM states. All these constructs are used to express our safety policy (no arithmetic overflow) and weakest preconditions for Jinja VM instructions. For simplicity this language is untyped. We do not even distinguish between formulas and expressions. Earlier instantiations [18] showed that this leads to duplication of functions and lemmas for both types. The uniform representation avoids this and still allows categorization with type checking functions. Next, we explain the semantics of all language constructs. Each expression can be evaluated for a given *jbc-state* and yields some Jinja value.

eval:: *jbc-state* $\Rightarrow$ *expr* $\Rightarrow$ *val*

The expressions come in two categories. From *Rg nat* to *Catch expr* we have JVM specific constructs. These are needed to access various parts of Jinja states

datatype *expr* =

Rg nat	register
| *St nat*	operand stack cell
| *Cn val*	constant value
| *NewA nat*	address for nth new object
| *Gf vname cname expr*	get field value
| *Ty expr ty*	type check
| *FrNr*	height of method frame stack
| *Pos pos*	position check
| *Call expr*	evaluation in call state
| *Catch cname expr*	evaluation in catch state
| *expr* ⌞+⌟ *expr* | *expr* ⌞−⌟ *expr* | *expr* ⌞*⌟ *expr*	arithmetic
| ⌞*if* *expr* ⌞*then*⌟ *expr* ⌞*else*⌟ *expr*	conditional
| *expr* ⌞=⌟ *expr* | *expr* ⌞≤⌟ *expr* | *expr* ⌞<⌟ *expr*	relational
| ⌞¬⌟ *expr*	negation
| *expr* ⌞⇒⌟ *expr*	implication
| ⌞⋀⌟ *expr list*	conjunction
| *Lv nat*	logical variable
| ⌞∀⌟ *nat expr*	quantification (logical vars)

Fig. 1. Jinja bytecode assertion language

in annotations. The remaining constructs are purely logical and are required to construct verification conditions.

3.1 JVM Specific Constructs

Since we want to use the annotation language to abstract Jinja states, we need various constructs to access different parts of such states. Most instructions only manipulate the topmost method frame. Instead of making the whole frame stack accessible in the language, which would complicate the evaluation range, we decided to use constructs for individual parts only. With *Rg k* and *St k* we access the *k*th register or element on the operand stack.

evalE Π (*p*,σ,*e*) (*Rg k*) = (*let* (*x*,*h*,*fs*)=σ; (*st*,*rg*,*p*)=*hd fs in rg*!*k*)
evalE Π (*p*,σ,*e*) (*St k*) = (*let* (*x*,*h*,*fs*)=σ; (*st*,*rg*,*p*)=*hd fs in st*!*k*)

Constants *Cn v* evaluate to their values *v*, i.e. *evalE Π s* (*Cn v*) = *v*. For better readability we abbreviate constants like *Cn* (*Intg 5*) or *Cn* (*Bool True*) with ⌞5⌟ or ⌞*True*⌟.

The *NewA n* expression returns the reference that is allocated in the heap for the *n*th object. In many cases this operator can be avoided as [5] shows. In our *wpF* operation, we will also replace field accesses of newly created objects by their default values. Eliminating all instances of *NewA* could be achieved at the level of formulas. However, this involves complex transformations of formulas and should be delegated to a post-processing function. In the definition we use the auxiliary function *new-Addr h*, which yields either *Some a* if *a* is the reference that is allocated next, or *None* if the heap is full.

evalE Π (p,σ,e) (NewA n) = (let (x,h,frs)=σ in evalNewA h n)
evalNewA h 0 = (case new-Addr h of None ⇒ Null | Some a ⇒ Addr a)
evalNewA h (Suc n) = evalNewA (h(the (new-Addr h):= Some arbitrary)) n

To evaluate *Gf F C ex*, which corresponds to $(C)ex.F$ in Java, we first check whether *ex* evaluates to an address value. If so, we fetch the value of the corresponding object field, otherwise we return *Unit*.

evalE Π s (Gf F C ex) = (case (evalE Π s ex)
of Addr a⇒(let (p,σ,e)=s;(x,h,frs)=σ;(D,fs)=the (h a)
in the (fs (F,C))))
| - ⇒ Unit)

The evaluation of *FrNr*, *Pos p* and *Ty ex tp* is straightforward, hence we skip the formal definitions. The *FrNr* expression yields the length of the method frame stack and *Pos p* evaluates to $\lfloor True \rfloor$ only if the current program position is *p*. To check the exact type of some expression we use *Ty ex tp*. Note that this check does not take the class hierarchy into account. Subtyping can be expressed by a disjunction of *Ty ex stp* expressions. With *Call* and *Catch* we evaluate formulas in previous states. The *Call ex* expression evaluates *ex* in the call state of the current method. This helps to specify postconditions of methods modularly. For example, annotating a `Return` instruction with $Rg\ 1 \sqsupseteq Call\ (Rg\ 1) \sqcap \lfloor 1 \rfloor$ means that the returning method has incremented register 1. This techniqe is related to primed variables in VDM [8], except that we can set entire expressions into a different temporal context, just like temporal logic operators do. This is important, as some method postconditions need old values of object fields or other parts of the heap. Another reason is that we use this operator to restore the call context when we compute the verification condition of a method return. For example, if register 1 had value *Intg 5* before the method call, the programmer might annotate the call position with $Rg\ 1 \sqsupseteq \lfloor 5 \rfloor$ and the return position with $Rg\ 1 \sqsupseteq \lfloor 6 \rfloor$. Our VCG would then produce the following proof obligation, where we have to show that the postcondtion together with the call annotation (evaluated in the call state!) imply the annotation at the return position: $(Rg\ 1 \sqsupseteq Call\ (Rg\ 1) \sqcap \lfloor 1 \rfloor \bigsqcap Call\ (Rg\ 1 \sqsupseteq \lfloor 5 \rfloor)) \sqsupseteq Rg\ 1 \sqsupseteq \lfloor 6 \rfloor$. Details about this construction can be found in [19]. We use the auxiliary function *call* to restore the call state of the current method. The program counter, registers and operand stack at call time are taken from the method frame beneath. The heap can be restored from the recordings in the environment. In case of a `main` method state (no caller), *Call ex* evaluates to *arbitrary*.

evalE Π s (Call ex) = (let (p,σ,e)=s; (x,h,frs)=σ in
(if length frs ≤ 1 then arbitrary else evalE Π (call s) ex))
call (p,(x,h,f#(s,r,p′)#frs),e) = (p′,(None,hd (cs e),(s,r,p′)#frs),e(|cs:=tl(cs e)|))

Exceptions impose a similar problem than method returns. However, since the number of frames chopped off the frame stack by exception handling is hard to determine statically, we need a special operator for this purpose. Just like *Call ex* the construct *Catch X ex* evaluates *ex* in a previous state. In this case we

restore the state under which we have last been in the `try` block that has a catching handler for exception X. The auxiliary function *catch* chops off frames until a catching handler is found. Simultaneously it restores the corresponding heap from the environment. Note that the resulting state is not the state under which the handler is entered, because the heap remains the same in this case.

$$\begin{array}{l} \mathit{evalE}\ \Pi\ s\ (\mathit{Catch}\ X\ ex) = (\mathit{let}\ (p,\sigma,e)=s;\ (x,h,frs)=\sigma\ \mathit{in} \\ \mathit{if}\ \mathit{length}\ frs \leq 1\ \mathit{then}\ \mathit{arbitrary}\ \mathit{else}\ \mathit{evalE}\ \Pi\ (\mathit{catch}\ (\mathit{fst}\ \Pi,X,s,ex))) \end{array}$$

$$\begin{array}{l} \mathit{catch}\ (P,X,(p,(x,h,fr\#(st,rg,p)\#frs),e)) = \\ (\mathit{let}\ (C,M,pc)=p\ \mathit{in}\ (\mathit{case}\ (\mathit{match\text{-}ex\text{-}table}\ P\ X\ pc\ (\mathit{ex\text{-}table\text{-}of}\ P\ C\ M)) \\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{catch}\ (P,X,(p,(\mathit{None},hd\ (cs\ e),(st,rg,p)\#frs),e(\!|cs{:=}tl(cs\ e)|\!))) \\ |\ \mathit{Some}\ pc' \Rightarrow (p,(\mathit{None},hd\ (cs\ e),(st,rg,p)\#frs),e(\!|cs{:=}tl\ (cs\ e)|\!)))) \end{array}$$

3.2 Logical Constructs

The arithmetic, relational, conditional and logical expressions are evaluated recursively. First, we evaluate the argument expressions, then we apply the corresponding arithmetic, relational or conditional operator on the results. If any argument value has not the expected type the result becomes *arbitrary*. A logical expression ex is true if it evaluates to *Bool True*, otherwise it is false. From Winskel [20] we take the idea of distinguishing program and logical variables. The first (registers, stack ...) depend on the *jvm-state* and may be modified by instructions. The latter are evaluated in a separate binding *lv e*, which we made part of the environment e in *jbc-state*, and are unaffected by instructions. In the substitutions we use later on to express the effect of instructions, we will neither transform nor introduce logical variables. Hence, no renaming of bound variables is required.

$$\mathit{evalE}\ \Pi\ (p,\sigma,e)\ (\mathit{Lv}\ k) = (\mathit{lv}\ e)\ k$$

Quantification only binds logical variables. The formula $\llbracket\forall\rrbracket\ v.\ ex$ holds, if ex holds no matter what value v' the logical variable $Lv\ v$ is bound to.

$$\begin{array}{l} \mathit{evalE}\ \Pi\ (p,\sigma,e)\ (\llbracket\forall\rrbracket\ v\ ex) = \\ \mathit{Bool}\ (\forall\, v'.\ \mathit{the\text{-}Bool}\ (\mathit{evalE}\ \Pi\ (p,\sigma,e(\!|lv{:=}((lv\ e)(v{:=}v'))|\!))\ ex)) \end{array}$$

3.3 Validity and Provability

To use this expresssion language as a logic, we need judgements for validity and provability. Models of formulas are program states under which a formula evaluates to *Bool True*.

$$\Pi,s \models ex = \mathit{the\text{-}Bool}\ (\mathit{evalE}\ \Pi\ s\ ex)$$

Provabiliy $\vdash$ is usually defined by giving a set of axioms and inference rules. However, we can also define provability semantically and use the inference system of HOL for proofs. We regard a formula as provable if we can prove in HOL that it holds for all states in the safety closure *safeP* Π of a program Π. This set of states is defined relative to some safety policy *safeF*, which we are going to instantiate in the next section.

$\Pi \vdash ex = \forall\ s{\in}\ safeP\ \Pi.\ \Pi{,}s \models ex$

The safety closure is defined inductively: All initial states are in *safeP* Π. If a state $(p{,}\sigma{,}e)$ is in *safeP* Π, satisfies the safety formula and annotation at p (if there is any), then every successor state $(p'{,}\sigma'{,}e')$, i.e. $((p{,}\sigma{,}e){,}(p'{,}\sigma'{,}e')) \in (effS\ \Pi)$, that satisfies the safety formula and annotation at p' is also in *safeP* Π.

4 Safety Policy

The assertion language serves three purposes: First, it provides a means to specify machine states and thus to annotate programs. Second, we use it to express verification conditions. Third, we use it to specify the safety policy. Our PCC framework expects the safety policy to be given as a function *safeF*:: *jbc-prog* $\Rightarrow$ *pos* $\Rightarrow$ *expr*, which defines a safety formula for each position p in a given program Π. This formula expresses conditions that must hold whenever we reach p at runtime. In our case we instantiate a safety policy that prohibits arithmetic overflow. The result of *IAdd* must not exceed *MAXINT*, which stands for Java's highest 32 bit integer $\lfloor 2147483647 \rfloor$.

$$safeF\ \Pi\ p = (if\ cmd\ \Pi\ p = Some\ IAdd\ \ then\ St\ 0 \lfloor{+}\rfloor St\ 1 \lfloor{\leqslant}\rfloor MAXINT \ else\ \lfloor True \rfloor)$$

A safety policy can also be lifted to programs. A **program is safe** if and only if every reachable state satisfies its safety formula.

$$isSafe\ \Pi = (\forall\, p_0\ \sigma_0\ e_0\ p\ m\ e.\ (p_0{,}\sigma_0{,}e_0) \in (initS\ \Pi) \wedge ((p_0{,}\sigma_0{,}e_0){,}(p{,}\sigma{,}e)) \in (effS\ \Pi)^* \longrightarrow \Pi{,}(p{,}\sigma{,}e) \models safeF\ \Pi\ p)$$

5 Verification Conditions

Our generic VCG analyses an annotated control flow graph and produces a formula in the assertion language. Details are in [19], here we only sketch the idea. Assume position p in program Π is annotated with A and has successor p', annotated with A'. A branch condition B specifies when p' is accessible from p. The verification condition for Π would contain the following proof obligation, ensuring a safe transition from p to p'.

$$(safeF\ \Pi\ p \lfloor\bigwedge\rfloor A \lfloor\bigwedge\rfloor B) \lfloor\Rightarrow\rfloor wpF\ \Pi\ p\ p'\ (safeF\ \Pi\ p' \lfloor\bigwedge\rfloor A')$$

We have to show that the safety formula at p, the annotation A and the branch condition B imply the weakest precondition for the safety formula and annotation at p'. The entire verification condition consists of various parts of this form. Not all positions must be annotated. It suffices if there is at least one annotation in each loop. For non-annotated positions the VCG constructs proof obligations by pulling back annotations of further successors using the weakest precondition function *wpF* and the successor function *succsF*. Relying on requirements for the parameter functions, we show in the PCC framework that the VCG is correct

and complete. The correctness theorem says that if we can prove the verification condition of a wellformed program, then this program is safe at runtime.

theorem vcg-correct: $wf\ \Pi \wedge \Pi \vdash vcg\ \Pi \longrightarrow isSafe\ \Pi$

Wellformedness means that there are enough annotations in the program. Completeness means that each wellformed and safe program with valid annotations yields a provable verification condition.

theorem vcg-complete: $wf\ \Pi \wedge correctAn\ \Pi \wedge isSafe\ \Pi \longrightarrow \Pi \vdash vcg\ \Pi$

Annotations are valid if they hold whenever the corresponding position is reached at runtime. For our instantiation to Jinja bytecode, we have proven both theorems by showing all the requirements on the parameter functions. The hardest part is to show that control flow function and the weakest precondition operator work correctly and precisely enough.

5.1 Jinja Bytecode Control Flow

To determine the control flow our VCG requires the function *succsF*, which given a program position yields a list of all direct successors paired with branch conditions. These specify the situations when a successor is accessible. In the definition we use separate functions for normal and exceptional successors. The auxiliary function *addPos* augments the branch conditions with a position formula *Pos p*. This connects verification conditions with the program structure and allows to weave in other properties (system invariants) in a post-processing step.

$$\begin{aligned} &succsF\ \Pi\ p = (case\ cmd\ \Pi\ p\ of\ None \Rightarrow [] \\ &\qquad\qquad | \ Some\ c \Rightarrow addPos\ p\ (succsNrm\ \Pi\ p\ c\ @\ succsExpt\ \Pi\ p\ c)) \\ &addPos\ p\ ss = map\ (\lambda\ (p',B).\ (p', \bigsqcap [B,Pos\ p]))\ ss \end{aligned}$$

The function *succsNrm* yields the successors for normal execution. For example *IfIntLeq* has two successors depending on whether the topmost stack entry *St 0* is less than or equal to *St 1*, the element beneath it.

$$\begin{aligned} &succsNrm\ \Pi\ (C,M,pc)\ (\texttt{IfIntLeq}\ t) = [((C,M,pc+t), St\ 0 \trianglelefteq St\ 1), \\ &\qquad\qquad\qquad\qquad\qquad\qquad ((C,M,pc+1), \sqsupset\ (St\ 0 \trianglelefteq St\ 1))] \end{aligned}$$

For instructions that might throw exceptions, *succsNrm* produces a branch condition that excludes this exception. We write $incA\ \ (C,M,pc)$ to increment positions, e.g. $(C,M,pc+1)$. The auxiliary function *xcpt-cond* generates a condition that ensures a particular exception.

$$\begin{aligned} &succsNrm\ \Pi\ p\ (\texttt{Getfield}\ F\ C\ ex) = [(incA\ p, \sqsupset (xcpt\text{-}cond\ \Pi\ NullPointer\ p))] \\ &cmd\ \Pi\ p = Some\ (\texttt{Getfield}\ F\ C) \longrightarrow xpct\text{-}cond\ \Pi\ X\ p = St\ 0 \sqsupseteq \lfloor Null \rfloor \end{aligned}$$

For `Putfield`, `New` and `Checkcast` the normal successors are determined analogously, only the type of exception differs. Method calls are more complicated, because overwriting opens multiple possibilities. It is hard to determine

statically the real type of the object whose method we are calling. However, we can ask the programmer or compiler to insert proper type annotations. Then we can select the corresponding method entry positions and construct sharp branch conditions.

succsNrm $\Pi\ p$ (`Invoke` $M\ n$) = *succsInvoke* (Π,M,n,p)

First, *succsInvoke* analyses the annotation at p to find out the types of the object reference on top of the stack. It expects this information to be given in form of a disjunction of *Ty ex tp* expressions. Then, it constructs exclusive branch conditions for each type.

succsInvoke (Π,M,n,p) = (*case anF* $\Pi\ p$ *of None* ⇒ []
| *Some A* ⇒ *concat* (*map* (λ *tp*. (*case tp*
of Class X ⇒ [((*X*,*M*,*0*),⌞¬⌟(*xcpt-cond* Π *NullPointer p*) ⌞⋀⌟ *Ty* (*St n*) (*Class X*))]
| - ⇒ [])) (*extractTy* (*A*,*St n*))))

For `Return` instructions we scan the code for all positions from which the current method could have been called. The name and class of the current method can be obtained from the position, say (C,M,pc), of the `Return` instruction. Then we scan the code for all positions p' with `Invoke` $M\ n$, which have *Ty* (*St n*) C in its annotation. For each of those call positions p' we construct a branch condition with the annotation, safety formula and position information of p'.

succsNrm $\Pi\ p$ `Return` =
map (λ p'. (*incA* p',*Call* (*And* [*assert* $\Pi\ p'$,*Pos* p']))) (*callers* $\Pi\ p$)
assert $\Pi\ p \equiv$ ⌞⋀⌟ [*safeF* $\Pi\ p$]@(*case anF* $\Pi\ p$ *of None* ⇒[] | *Some A* ⇒ [*A*])

For the remaining instructions *succsNrm* can be defined analogously to the shown examples. When instructions throw exceptions control flows to an appropriate handler. The function *succsExpt* checks which exceptions each instruction may throw and invokes *succsXpt* to find potential handlers. Example:

succsExpt $\Pi\ p$ (`Getfield` $F\ C$) = *succsXpt* (Π,*NullPointer*,[p])

Handlers are searched by recursively climbing up the call tree and inspecting the exception tables of each call method. In *succsXpt* we keep a list of visited positions. When this list becomes too long or empty, *succsXpt* terminates by making all program positions potential successors. This means programs with uncaught exceptions usually yield unprovable verification conditions. However, adding a global exception handler to the main method always helps to avoid this problem. When *succsXpt* finds a handler it constructs a branch condition that specifies under which situation this handler is selected. When an exception is caught in the same method as it is thrown (pss = []), we get branch condition ⌞*True*⌟, otherwise we restore the call context using *Catch* on the annotation and safety formula of the call point.

succsXpt (Π,X,ps) = (*if length* (*domC* Π) ≤ *length ps* ∨ ps=[]
then map (λ*p*. (p,⌞*True*⌟)) (*domC* Π)
else (*let p*=*fst ps*; (*C*,*M*,*pc*)=*p*; *et*=*ex-table-of P C M*; *A*=*assert* $\Pi\ p$
in (*case match-ex-table P X pc et*

of None ⇒ *concat* (*map* (λ*p′*. *succsXpt* (*Π*,*X*,*p′*#*ps*)) (*callers Π p*))
Some h ⇒ [((*C*,*M*,*h*), ⋀ (*if pss*=[] *then* [] *else* [*Catch X A*])@[*xcpt-cond Π X* (*last ps*)])]

5.2 Weakest Preconditions

The purpose of the *wpF* operator is to express the semantics of the underlying programming language at the level of formulas. We have proven that our *wpF* operator satisfies the following lemma, which implies the requirements we need for correctness and completeness of our *VCG*.

lemma *wf Π* ∧ (*p*,*m*,*e*)∈ *safeP Π* ∧ ((*p*, *m*, *e*), (*p′*, *m′*,*e′*)) ∈ *effS Π*
⟶ *evalE Π* (*p*,*m*,*e*) (*wpF Π p p′ Q*) = *evalE Π* (*p′*, *m′*,*e′*) *Q*

Roughly speaking this lemma says that *wpF Π p p′ Q* transforms a postcondition *Q* such that it evaluates to the same value as *Q* does in the successor state. This can be done by substituting all expressions of a formula *Q* that change its value due to the effect of an instruction by another expressions that yields the same value in the predecessor state. The substitution function *substE*::(*expr* ▷ *expr*) ⇒ *expr* ⇒ *expr* maintains an expression map *em*. It traverses a given formula (not descending into temporal constructs) and simultaneously replaces all instances of expressions that appear on the left hand side of a maplet in *em* by the corresponding right hand side. Example:

substE [(*St 0*,*Rg 0*)] (*St 0* ⌞⌟ *Call* (*St 0*)) = *Rg 0* ⌞⌟ *Call* (*St 0*)

Usually substitution only replaces variables. However, Jinja Bytecode instructions may also change the heap. Hence, we sometimes have to substitute entire expressions. In the definition of *wpF* we analyse the postcondition and extract subexpressions of particular patterns. These are then used to build maplets for the substitution map.

wpF Π p p′ Q = (*let pm*=*map* (λ*q*. (*Pos q*,*q*=*p′*⌟)) (*getPosEx Q*)
in (*case cmd Π p of None* ⇒ *FF* | *Some ins* ⇒ (*case handlesEx* (*fst Π*) *p′*
of None ⇒ *wpFNrm Π p p′ Q pm ins* | *Some cn* ⇒ *wpFExpt Π p p′ Q pm ins*)))

When evaluating *wpF Π p p′ Q* we assume that the program counter changes from *p* to *p′*. This means we can eliminate position expressions *Pos q* in *Q*, which we extract with *getPosEx Q*. If *p′* is the start address of some handler for an exception *cn*, we assume that the transition from *p* to *p′* is due to an exception and delegate work to *wpFExpt*. Otherwise we use *wpFNrm*, which transforms *Q* according to normal execution. For example in case of an *IAdd* instruction *wpFNrm* replaces instances of *St 0* with *St 0* ⌞+⌟ *St 1*, which has the same value as *St 0* has in the successor state. Since *IAdd* reduces the stack, all other instances of *St k*, whose indexes are extracted by *stkIds*, get shifted.

wpFNrm Π p p′ pm IAdd = *substE* (*pm*@
(*map* (λ*k*. (*St k*,*if k*=*0 then St 0* ⌞+⌟ *St 1 else St* (*k*+*1*))) (*stkIds Q*))) *Q*

In case of `Getfield` *F C* we only have to transform *St 0*.

$wpFNrm\ \Pi\ p\ p'\ pm\ (Getfield\ F\ C) \equiv substE\ (pm@[(St\ 0, Gf\ F\ C\ (St\ 0))])\ Q$

Like other instructions that affect the heap, `Putfield` is more tricky to handle. Apart from shifting the stack, which gets reduced by two elements, we have to scan Q for all expressions that depend on the heap. These are all expressions of the form $Gf\ F\ C\ ex$, which we extract in subterm order with *getGfEx*. For each instance, we first transform ex using the maplets we have found so far. Then we build an expression that checks whether ex' equals the reference of the changed object (in *St 1*). If yes, we replace $Gf\ F\ C\ ex$ with the new object field value, stored in *St 0*. Otherwise, we take the transformed version $Gf\ F\ C\ ex'$.

$$
\begin{array}{l}
wpFNrm\ \Pi\ p\ p'\ Q\ pm\ (Putfield\ F\ C) = (\\
let\ em{=}pm@(map\ (\lambda k.\ (St\ k, St\ (k{+}2)))\ (stkIds\ Q));\\
\quad gfe'{=}foldl\ (\lambda mp\ ex.\ let\ x{=}substE\ mp\ ex\\
\qquad\qquad\qquad\qquad in\ (Gf\ F\ C\ ex, IF\ x_{\sqsupset} St\ 1\ THEN\ St\ 0\ ELSE\ Gf\ F\ C\ x)\#mp)\\
\qquad\quad em\ (getGfEx\ F\ C\ Q)\\
in\ substE\ gfe'\ Q)
\end{array}
$$

Another tricky instruction is `Invoke` $M\ n$. Since the successor state has one frame more, we replace $FrNr$ with $FrNr \sqcup 1$. Since the call state of the successor state is the current state we replace instances of $Call\ ex$ with ex. In case of $Catch\ X\ ex$ we check wether the current method has an appropriate handler. If so, we can eliminate the *Catch*. Otherwise, we replace it with an expression that checks wether the current state only has one frame. In this case evaluation of $Catch\ X\ ex$ equals ex, hence we eliminate *Catch* again. Otherwise, we leave it. Finally, we handle the argument passing. The first n elements of the stack are written into the registers *1* to $n{+}1$ in reversed order. We create maplets that substitute each register with the corresponding stack position of the predecessor state. The stack is emptied, which amounts to replacing all references $St\ k$ with $\lfloor arbitrary \rfloor$.

$$
\begin{array}{l}
wpFNrm\ \Pi\ p\ p'\ Q\ pm\ (\texttt{Invoke}\ M\ n) = substE\ (pm@(FrNr, FrNr \sqcup Cn\ (Intg\ 1))\#\\
(map\ (\lambda k.\ (Rg\ k, if\ k \le n\ then\ St\ (n{-}k)\ else \lfloor arbitrary \rfloor))\ (rgIds\ Q))@\\
(map\ (\lambda k.\ (St\ k, \lfloor arbitrary \rfloor))\ (stkIds\ Q))@\\
(map\ (\lambda x.\ (Call\ x, x))\ (getCallEx\ Q))@\\
(concat\ (map\ (\lambda(c,x).(if\ catchesEx\ (fst\ \Pi)\ c\ p\ then\ [(Catch\ c\ x, x)]\\
\qquad\qquad\qquad\qquad else\ [(Catch\ c\ x, IF\ FrNr_{\sqsupset} 1\ THEN\ x\\
\qquad\qquad\qquad\qquad\quad ELSE\ Catch\ c\ x)]))\ (getCatchEx\ Q)))\ Q
\end{array}
$$

In case of `Return` the successor state has one frame less. Hence, the evaluation of *Call* and *Catch* expressions needs to be adjusted again. Adding an additional *Call* to such expressions amounts to the same as chopping off the topmost frame of the current state. We skip the formal defintion for *Return* and the remaining instructions, as the same techniques apply as before. It turns out that the instructions that affect the heap or the structure of the frame stack are significantly more difficult to handle. Exception handling works similar for all instructions, but `Throw`, which needs to be treated special because we do not know from the code which exception is thrown. Similarly to *Invoke* $M\ n$ we require that potential classes are annotated in form of $Ty\ ex\ tp$ expressions.

```
wpFExpt Π p p' Q pm cn = (let mp=pm@(map (λk. (St k,if 1≤k then ⌞arbitrary⌟
else (if (cmd Π p = Some Throw)
then (IF St 0 ≔ ⌞Null⌟ THEN ⌞addr-of-sys-xcpt NullPointer⌟ ELSE St 0)
else ⌞addr-of-sys-xcpt cn⌟))) (stkIds Q))@
(let (C,M,pc)=p; (C',M',pc')=p'; (P,An)=Π
in (if match-ex-table P cn pc (ex-table-of P C M) = Some pc' then [] else
    let rgm=map (λk. (Rg k,Catch cn (Rg k))) (rgIds Q);
        om=map (λex. (Call ex,Catch cn (Call ex))) (getCallEx Q);
        cm=map (λ(c,x). (Catch c x, Catch cn (Catch c x))) (getCatchEx Q)
        in (FrNr,Catch cn FrNr)#rgm@om@cm))
in substE mp Q)
```

6 Example Program

This section presents a small example program, which we have proven safe.

```
Cl ::jvm-method cdecl
Cl ≡ (A,                    class A {
(Object,[(n,Integer)],[     int n;
(sum,[],Integer,(2,2,[      int sum() {
Push (Intg 0) − ''pre'',    {int k = 0;
Store k,
Push (Intg 0),                int r = 0;
Store r,
Load k,
Load this − ''inv'',
Getfield n A,
IfIntLeq 10,                while (k < n)
Load k,                       {
Push (Intg 1),
IAdd,
Store k,                      k = k + 1;
Load r,
Load k,
IAdd,
Store r,                      r = r + k;
Goto −12                      }
Load r,
Return − ''post'' ,           return r; }
],[]))]))                   }
```

Fig. 2. Example Program

Method *sum* in class *A* adds the numbers from *0* to field value *n*. To verify this program we need annotations. The precondition says field *n* has not changed since call time and ranges between *0* and *65535*, the highest input for which the sum does not overflow.

$pre \equiv \bigwedge [Rg\ 0 \coloneqq Call\ (St\ 0),\ Gf\ n\ A\ Rg\ 0 \coloneqq Call\ (Gf\ n\ A\ St\ 0),\ Gf\ n\ A\ Rg\ 0 \leqslant \lfloor 65535 \rfloor,\ \lfloor 0 \rfloor \leqslant Gf\ n\ A\ Rg\ 0]$

The invariant contains type restrictions and the Gaussian summation formula. It also says that *n* does not change and that *k* ranges between *Intg 0* and the value of *n*.

$inv \equiv \bigwedge [Ty\ Rg\ k\ Integer,\ Ty\ Rg\ r\ Integer,\ \lfloor 2 \rfloor \lfloor * \rfloor Rg\ r \coloneqq Rg\ k \lfloor * \rfloor (Rg\ k \lfloor + \rfloor \lfloor 1 \rfloor),\ Gf\ n\ A\ Rg\ 0 \coloneqq Call\ (Gf\ n\ A\ St\ 0),\ Gf\ n\ A\ Rg\ 0 \leqslant \lfloor 65535 \rfloor,\ \lfloor 0 \rfloor \leqslant Gf\ n\ A\ Rg\ 0,\ Rg\ k \leqslant Gf\ n\ A\ Rg\ 0,\ \lfloor 0 \rfloor \leqslant Rg\ k]$

The postcondition contains type information again, and a formula that relates the result value to the input value *n*, which still has the same value as at call time.

$post \equiv \bigwedge [St\ 0 \coloneqq Rg\ r,\ Ty\ Rg\ r\ Integer,\ Gf\ n\ A\ Rg\ 0 \coloneqq Call\ (Gf\ n\ A\ St\ 0),\ \lfloor 2 \rfloor \lfloor * \rfloor Rg\ r \coloneqq Call\ (Gf\ n\ A\ St\ 0) \lfloor * \rfloor (Call\ (Gf\ n\ A\ St\ 0)) \lfloor + \rfloor \lfloor 1 \rfloor]$

Proving the verification condition of this program is automatic using a tactic for bounded arithmetics. The type annotations are used to trigger simplification rules that translate the arithmetic expressions of type *expr* turning up in the verification conditions to arithmetic expressions in Isabelle/HOL. For the latter powerful decision procedures, such as Presburger Arithmetics are available. We have tested Jinja programs that call this method up to the size of *10.000* instructions. Up to this size the decision procedures and the simplifier scale pretty well. Details can be found online [1].

7 Generating Annotations

The annotations in the example program have been added manually. They have been designed to make the verification go through automatically with a general setup of simplification rules and decision procedures. In practice a fully automatic approach would be desirable. In many cases program analysis can help to find proper annotations. For example the type information of the annotations above could be directly transferred from the bytecode verifier's analysis. To find out upper and lower bounds of expressions, we can employ interval analysis for Java bytecode [17]. More complex invariants could be gained by advanced analysis techniques, such as polyhedra or affine relations [12]. For annotations involving analysis of pointer structures TVLA [11] can help. Since the generation of annotations need not be trusted, a wide range of options are available at this point.

8 Conclusion

To our knowledge the literature on Java does not propose a logic to annotate and verify bytecode. In this paper we tried to fill this gap for a safety policy against arithmetic overflow and a bytecode subset that covers the essential object oriented features. This and various other instantiations of our PCC framework [1] show that a PCC system with formally verified trusted components is feasible. The infrastructure an interactive theorem prover like Isabelle/HOL provides is very useful. In particular the ability to generate proofs with decision procedures or interactively with the full power of HOL available, turns out to be a good strategy in a field where Rice's theorem shatters the dream of complete automation.

References

1. VeryPCC project website in Munich, http://isabelle.in.tum.de/verypcc/, 2003.
2. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In *Verification: Theory and Practice*, volume 2772 of *Lect. Notes in Comp. Sci.*, pages 11–41. Springer-Verlag, 2004.
3. A. W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258, June 2001.

4. D. Aspinall, L. Beringer, M. Hoffman, and H.-W. Loidl. A resource-aware program logic for a jvm-like language. In S. Gilmore, editor, *Trends in Functional Programming*. Edinburgh, 2003.
5. F. D. Boer and C. Pierik. A syntax-directed hoare logic for object-oriented programming concepts. In *Proceedings of Formal Methods for Open Object-based Distributed Systems (FMOODS)*, LNCS. Springer, 2003.
6. J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound tal for back-end optimization. In *Programming Languages Design and Implementation (PLDI)*. ACM Sigplan, 2003.
7. D. L. Detlefs, K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical report, Compaq Systems Research Center, 1998.
8. C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 2nd edition, 1990.
9. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Research report, National ICT Australia, Sydney, 2004.
10. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. Jml reference manual (draft). Technical report, 2004.
11. T. Lev-Ami, T. Reps, M. Sagiv, and T. Wilhelm. Putting static analysis to work for verification: A case study in issta 2000. Technical report, 2000.
12. M. Mueller-Olm and H. Seidl. Program analysis through linear algebra. In *31st Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 330–341, 2004.
13. G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.
14. T. Nipkow and A. Chaieb. Generic proof synthesis for presburger arithmetic – draft. Technical report, Technische Universitaet Muenchen, 2004.
15. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer, 2002.
16. D. v. Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.
17. M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. *Proceedings of the 1st Workshop on Bytecode (Bytecode05)*, 2005. submitted for publication.
18. M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In *Proc. 17th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs 2004)*. Springer Verlag, 2004. 16 pages.
19. M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In *Proc. 3rd IFIP Int. Conf. Theoretical Computer Science (TCS 2004)*, 2004.
20. G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

Subtyping First-Class Polymorphic Components

João Costa Seco and Luís Caires

Departamento de Informática,
Universidade Nova de Lisboa
{Joao.Seco, Luis.Caires}@di.fct.unl.pt

Abstract. We present a statically typed, class-based object oriented language where classes are first class polymorphic values. A main contribution of this work is the design of a type system that combines first class polymorphic values with structural equirecursive types and admits a subtyping algorithm which is arguably much simpler than existing alternatives. Our development is modular and can be easily instantiated for either a Kernel-Fun or a $F^{\top}_{\leq}$ style of subtyping discipline.

1 Introduction

When one considers the essence of programming languages and especially of programming languages with subtyping and equirecursive types the language that comes to mind is $F_{\leq}$, the extension with subtypes of the polymorphic lambda calculus introduced in [7]. When object-oriented programming is at stake, languages like the object calculus of Abadi and Cardelli [1] and Momi [15] are good examples of how to express object-oriented mechanisms. However, the subtyping relations they use to relate classes, objects and mixins stays far behind the flexible relation of $F_{\leq}$; both use invariant width subtyping relations and limited recursive subtyping relations. This expressiveness gap is due to unsoundness problems when coding the *self* reference as a generic value parameter of methods. In FJ [13], for instance, structural equivalence of types is traded by name-based equivalence, which is convenient to overcome problems with the subtyping of recursive types. In fact, structural equivalence of types, although adopted in some experimental programming languages such as OCaml and Modula3, does not seem to have had substantial impact in main-stream object-oriented languages.

Nevertheless, the increasing use of dynamic loading, late binding and mobile code in general purpose programming frameworks raises the issue of finding more flexible compatibility criteria between components. One reason is that name-based extension and subtyping as it is implemented by modern object oriented languages creates a rigid hierarchy of classes and interfaces based on their names. This implies the usage of global name spaces and, for instance, disallows the compatibility of two classes that separately combine the same set of interfaces. This problem can of course be diminished by explicitly using wrapper objects that redirect method calls and therefore make compatible two otherwise incompatible classes. But, structural equivalence would be the most natural solution to this kind of problems.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 342–356, 2005.

In previous work we have presented an object oriented component calculus that uses structural equivalence of types [18, 19]. The calculus was ported into an experimental language that is compiled to run on the Java framework, which inherits the structural character of calculus' type relations up to a certain level. In this paper, we extract the essential aspects of the type system of the component calculus into a core class-based language that manipulates classes as values and uses second-order equirecursive types.

One of the main contributions of this paper is the presentation of a subtyping algorithm for second-order equirecursive types. Our approach is intuitive and technically much simpler to define and prove correct than previous results, such as [5]. It builds on the coinductive formulation of first-order type systems with equirecursive types of Amadio and Cardelli [2], Brandt and Henglein [3], and Gapeyev, Levin and Pierce [9, 17]. The simplicity of the algorithm is directly related to the nature of the elements the algorithm manipulates which are complete judgments instead of pairs of types and to the termination conditions of the coinductive algorithm, which uses permutation-based techniques. Moreover, our development is modular, in the sense that it can be adapted to either a Kernel-Fun or a $F^{\top}_{\leq}$ subtyping discipline.

A further contribution is of this paper is the proposal of a simple composition mechanism for classes that combines the usage of structural equivalence of classes and objects with class extension mechanisms.

The remainder of the paper is structured as follows: section 2 describes the class-based language and illustrates it using a small programming example; in section 3 we define a type system for the language including the subtyping relation and its properties; in section 4 we describe the algorithm that checks the type of a language expression and enunciate its properties. Finally, in section 5 we propose a new composition mechanism for extending classes that is sound under structural subtyping of classes and objects.

2 The Programming Language

In this section, we define a core class-based programming language whose values are classes and objects. Both objects and classes are runtime entities in our language, the main goal is to study a type system for generic components (as classes) and objects, where the polymorphic types of classes and equirecursive types of objects are related structurally. We first introduce its syntax, which is depicted in Fig. 1. It includes constructs for objects, classes, instantiation of objects, method calls, local declarations, and recursion. Class expression combines bounded type abstraction, and value abstraction over a name denoting the object *self*. All other constructions are interpreted as shown in Fig. 2.

The evaluation relation of the language is defined by a big step semantics $e \Downarrow v$. Among these rules the evaluation of a $\mathsf{new}\ e$ expression deserves further explanation: it relies on the evaluation of the subexpression e into a class value, and closure under recursion of $self$. All other rules evaluate the corresponding

Types

$$
\begin{array}{lll}
\tau ::= X & & \text{(variable)} \\
\quad \mid \ \mathsf{Class}[X_i \leq \tau_i \ ^{i\in 1..n}]\ I & & \text{(class)} \\
\quad \mid \ I & & \text{(interface)} \\
\quad \mid \ \mathsf{Top} & & \text{(top)} \\
I ::= \{m_i(\tau_{j_i} \ ^{j_i\in 1..n_i}) : \tau_i \ ^{i\in 1..n}\} & & \text{(interface)} \\
\quad \mid \ \mu X.I & & \text{(recursion)}
\end{array}
$$

Terms

$$
\begin{array}{lll}
e ::= x & & \text{(variable)} \\
\quad \mid \ v & & \text{(value)} \\
\quad \mid \ \mathsf{new}\ e[\tau_i \ ^{i\in 1..n}] & & \text{(instantiation)} \\
\quad \mid \ e.m(e_i \ ^{i\in 1..n}) & & \text{(method call)} \\
\quad \mid \ \mathsf{let}\ x = e\ \mathsf{in}\ e & & \text{(declaration)} \\
\quad \mid \ \mathsf{rec}(x:\tau)\ e & & \text{(recursion)}
\end{array}
$$

Values

$$
\begin{array}{lll}
v ::= \{\ m_i(x_{j_i} : \tau_{j_i} \ ^{j_i\in 1..n_i}) : \tau_i = e_i \ ^{i\in 1..n}\ \} & & \text{(objects)} \\
\quad \mid \ \mathsf{class}[X_i \leq \tau_i \ ^{i\in 1..n}](s)\ e & & \text{(classes)}
\end{array}
$$

Fig. 1. Types and terms

$$v \Downarrow v \ \text{(Value)} \qquad \frac{e_1 \Downarrow v_1 \quad e_2[x \leftarrow v_1] \Downarrow v_2}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow v_2}\ \text{(Let)} \qquad \frac{e[x \leftarrow \mathsf{rec}(x:\tau)\ e] \Downarrow v}{\mathsf{rec}(x:\tau)\ e \Downarrow v}\ \text{(Fix)}$$

$$\frac{\begin{pmatrix} o = \{m_i(x_{j_i} : \tau_{j_i} \ ^{j_i\in 1..n_i}) : \tau_i = e_i \ ^{i\in 1..n}\} \\ I = \{m(x_{j_i} : \tau_{j_i} \ ^{j_i\in 1..n_i}) : \tau_i \ ^{i\in 1..n}\} \end{pmatrix} \quad e \Downarrow \mathsf{class}[X_i \leq \delta_i \ ^{i\in 1..n}](s)\ o \qquad \mathsf{rec}(s:I)\ o \Downarrow v}{\mathsf{new}\ e[\tau_i \ ^{i\in 1..n}] \Downarrow v}\ \text{(New)}$$

$$\frac{e \Downarrow \{\ \ldots, m(x_i : \tau_i \ ^{i\in 1..n}) : \tau = e_b, \ldots\ \} \quad e'_i \Downarrow v_i \ \ \forall_{i\in 1..n} \quad e_b[x_i \leftarrow v_i \ ^{i\in 1..n}] \Downarrow v}{e.m(e'_i \ ^{i\in 1..n}) \Downarrow v}\ \text{(Call)}$$

Fig. 2. Big step operational semantic rules

expressions as expected. Note that classes and objects are values and hence evaluate to themselves by means of the rule (Value).

Types already appear in the syntax of expressions, are are also defined in Fig. 1. We distinguish between two kinds of types: interface types and class types. Type variables range over types of any kind. A class type $\mathsf{Class}[X \leq \tau]\ \sigma$ is a polymorphic type corresponding to a $F_{\leq}$ bounded type quantification (cf., $\forall_{X\leq\tau}.\sigma$), but for convenience generalized to a list of type parameters. Interface types can be defined recursively, using type recursion $\mu X.I$; our separation of types in two categories τ and I is not essential, and only reflects the intended type usage of the object-oriented language.

We illustrate our language with a very simple example that uses polymorphic memory cells. Let `C` be the type defined as $\mathsf{Class}[X]\ \mu Y.\{set(X) : Y,\ get() : X\}$, and `cell` some class value of type `C`, and consider

$$\phi \vdash \diamond \ \text{(E-}\phi\text{)} \qquad \frac{\Delta \vdash \tau \ \mathsf{ok}}{\Delta, x : \tau \vdash \diamond} \ \text{(E-Var)} \qquad \frac{\Delta \vdash \tau \ \mathsf{ok}}{\Delta, X \leq \tau \vdash \diamond} \ \text{(E-TVar)}$$

$$\frac{\Delta \vdash \diamond}{\Delta \vdash X \ \mathsf{ok}} \ \text{(O-TVar)} \qquad \frac{\Delta, X \leq \mathsf{Top} \vdash I \ \mathsf{ok}}{\Delta \vdash \mu X.I \ \mathsf{ok}} \ \text{(O-Recursive)}$$

$$\frac{\Delta, X_j \leq \tau_j \ ^{j \in 1..i-1} \vdash \tau_i \ \mathsf{ok} \quad \forall_{i \in 1..n} \quad \Delta, X_i \leq \tau_i \ ^{i \in 1..n} \vdash I \ \mathsf{ok}}{\Delta \vdash \mathsf{Class}[X_i \leq \tau_i \ ^{i \in 1..n}] \ I \ \mathsf{ok}} \ \text{(O-Polymorphic)}$$

$$\frac{\Delta \vdash \tau_{j_i} \ \mathsf{ok} \ \ \forall_{j_i \in 1..n_i} \ \forall_{i \in 1..n} \quad \Delta \vdash \tau_i \ \mathsf{ok} \ \forall_{i \in 1..n}}{\Delta \vdash \{m_i(\tau_{j_i} \ ^{j_i \in 1..n_i}) : \tau_i \ ^{i \in 1..n}\} \ \mathsf{ok}} \ \text{(O-Interface)}$$

Fig. 3. Well-formed types and environments

```
let d = class[Y](s){
           test(c:C, v:Y):Y =
             let o1 = (new c[Y]) in
             let o2 = o1.set(v) in o2.get()
      }
  in let n = (new d[int]).test(cell,1)
```

Class `d` defines a method `test` that accepts two arguments: a class value (a component) implementing memory cells `c` and another appropriate value `v`. The method instantiates the memory cell class, stores the value `v` in the resulting cell object, and finally retrieves the cell contents and returns it.

3 Type System

In this section we define the type system for our language. The type system has two parts, a typing system for the language expressions, and a subtyping system expressing the intended subsumption relation on types. Our presentation will focus on the latter, concentrating on the development of our approach to polymorphic recursive subtyping.

3.1 Typing Expressions

The typing of the expressions is given by the set of rules in Fig. 4, that proves judgements of the form $\Delta \vdash e : \tau$, where Δ is the typing environment declaring the types of the free value and type variables relevant for the expression e, and τ is a type. Well-formed types and environments are also defined in Fig. 3, by the judgement forms $\Delta \vdash \diamond$ and $\Delta \vdash \tau \ \mathsf{ok}$, as expected.

Most rules follow the usual pattern, we will discuss just the particularities of our presentation. Although the abstract syntax in Fig. 1 is somewhat more liberal, notice that rule (T-Class) enforces that only record expressions are accepted as a class body, consistently with our interpretation of polymorphic types as polymorphic object-generating classes. The name s, whose scope is the class

$$\frac{x:\tau \in \Delta}{\Delta \vdash x:\tau}\ \text{(T-Var)} \qquad \frac{\Delta \vdash e:\tau' \quad \Delta \vdash \tau' \leq \tau}{\Delta \vdash e:\tau}\ \text{(T-Sub)} \qquad \frac{\Delta, x:\tau \vdash e:\tau}{\Delta \vdash \mathsf{rec}(x:\tau)\ e:\tau}\ \text{(T-Fix)}$$

$$\frac{\begin{pmatrix} c = \{m_i(x_{j_i}:\tau_{j_i}\ ^{j_i \in 1..n_i}) : \tau_i = e_i\ ^{i\in 1..n}\} \\ I = \{m_i(\tau_{j_i}\ ^{j_i\in 1..n_i}) : \tau_i\ ^{i\in 1..n}\} \end{pmatrix} \quad \Delta, X_j \leq \delta_j\ ^{j\in 1..m}, x_{j_i}:\tau_{j_i}\ ^{j_i\in 1..n_i}, s : I \vdash e_i : \tau_i\ \ \forall_{i\in 1..n}}{\Delta \vdash \mathsf{class}[X_j \leq \delta_j\ ^{j\in 1..m}](s)\ c : \mathsf{Class}[X_j \leq \delta_j\ ^{j\in 1..m}]\ I}\ \text{(T-Class)}$$

$$\frac{(\ I = \{m_i(\tau_{j_i}\ ^{j_i\in 1..n_i}) : \tau_i\ ^{i\in 1..n}\}\ ) \quad \Delta, x_{j_i}:\tau_{j_i}\ ^{j_i\in 1..n_i} \vdash e_i : \tau_i\ \ \forall_{i\in 1..n}}{\Delta \vdash \{\ m_i(x_{j_i}:\tau_{j_i}\ ^{j_i\in 1..n_i}) : \tau_i = e_i\ ^{i\in 1..n}\ \} : I}\ \text{(T-Object)}$$

$$\frac{\Delta \vdash e : \mathsf{Class}[X_i \leq \delta_i\ ^{i\in 1..n}]\ I \quad \Delta \vdash \tau_i \leq \delta_i[X_j \leftarrow \tau_j\ ^{j\in 1..i-1}]\ \ \forall_{i\in 1..n}}{\Delta \vdash \mathsf{new}\ e[\tau_i\ ^{i\in 1..n}] : I[X_i \leftarrow \tau_i\ ^{i\in 1..n}]}\ \text{(T-New)}$$

$$\frac{\Delta \vdash e : \{\ldots, m(\tau_i\ ^{i\in 1..n}) : \tau, \ldots\} \quad \Delta \vdash e_i : \tau_i \quad \forall_{i\in 1..n}}{\Delta \vdash e.m(e_i\ ^{i\in 1..n}) : \tau}\ \text{(T-Call)} \qquad \frac{\Delta \vdash e_1 : \tau \quad \Delta, x:\tau \vdash e_2 : \tau'}{\Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau'}\ \text{(T-Let)}$$

Fig. 4. Typing rules

body, typed with the type of the instances of the class, is our notation for *self*. In the rule (T-New), the instantiation expression new is typed with a type constructed from the type declarations present in the class expression, given the proper type substitution of the type parameters, and provided that the compatibility of the type arguments with respect to the variable bounds holds.

3.2 Subtyping

Some approaches to the problem of defining a first-order subtyping relation between recursive types exist for quite a while [2, 3, 17].

Intuitively, the intended subsumption relation between recursive types corresponds to the usual inclusion of infinite (regular) trees, the difficulty in the polymorphic case arises due to the presence of binding occurrences of type variables on types, due to presence of type quantifiers. Usually, even for recursive types, subtyping relations have been expressed by means of inductive proof systems, where the coinduction principle appears embedded in various explicit ways [2, 3]. Apart from these, the main proof rules we might expect for such a subtyping system are the ones depicted in Fig. 5. These include the usual relationships: maximality of Top, reflexivity, transitivity, width and depth record subtyping, and unfolding of recursive types. For comparing polymorphic types, we adopt a Kernel-Fun style rule, since the more general $F_{\leq}$ style subtyping is known to be undecidable even in the absence of recursion [16]. In any case, our approach applies equally well to Kernel-Fun and to variants such as $F^{\top}_{\leq}$.

Unfortunately, the adoption of these rules results in an incomplete type system, that does not seem to easily lead to a terminating algorithm, as remarked in [11], although a rather complex subtyping algorithm following this approach

$$\Delta \vdash \tau \leq \mathsf{Top} \qquad \Delta \vdash \tau \leq \tau \qquad \frac{\Delta \vdash \tau \leq \sigma \qquad X \leq \tau \in \Delta}{\Delta \vdash X \leq \sigma}$$

$$\frac{\Delta, X_i \leq \delta_i \ ^{i\in 1..n} \vdash I \leq I'}{\Delta \vdash \mathsf{Class}[X_i \leq \delta_i \ ^{i\in 1..n}]\ I \leq \mathsf{Class}[X_i \leq \delta_i \ ^{i\in 1..n}]\ I'}$$

$$\frac{n' \leq n \qquad \Delta \vdash \tau_i \leq \tau'_i \ \forall_{i\in 1..n'} \qquad \Delta \vdash \tau'_{j_i} \leq \tau_{j_i} \ \forall_{j_i \in 1..n_i} \forall_{i\in 1..n'}}{\Delta \vdash \{m_i(\tau_{j_i} \ ^{j_i\in 1..n_i}) : \tau_i \ ^{i\in 1..n}\} \leq \{m_i(\tau'_{j_i} \ ^{j_i\in 1..n_i}) : \tau'_i \ ^{i\in 1..n'}\}}$$

$$\frac{\Delta \vdash I \leq J[\alpha \leftarrow \mu\alpha.J]}{\Delta \vdash I \leq \mu\alpha.J} \qquad \frac{\Delta \vdash I[\alpha \leftarrow \mu\alpha.I] \leq J}{\Delta \vdash \mu\alpha.I \leq J}$$

Fig. 5. Subtyping inductive rules

was already developed in [5]. In fact, our difficulties in getting a clear understanding of this work lead us to attempt a different approach, leading to the presentation in this paper. We then follow essentially the development of [9] for first-order types, and extend it in a natural way to polymorphic types. Therefore, we start from a coinductive definition of the subtyping relation, presented below.

Let $\mathcal{D}$ denote the set of all well-formed environments and $\mathcal{T}$ the set of all well-formed types. We also denote by $\mathcal{J}$ the set $\mathcal{D} \times \mathcal{T} \times \mathcal{T}$ of all subtyping judgements (represented as tuples).

Definition 3.1. *The* subtyping generating function *is the mapping* $S \in \mathcal{P}(\mathcal{J}) \rightarrow \mathcal{P}(\mathcal{J})$ *defined by:*

$$\begin{aligned}
S(\mathfrak{R}) = {} & \{(\Delta; \tau; \tau) \mid \tau \in \mathcal{T} \text{ and } FV(\tau) \subseteq Dom(\Delta)\} \\
\cup {} & \{(\Delta; \tau; \mathsf{Top}) \mid \tau \in \mathcal{T} \text{ and } FV(\tau) \subseteq Dom(\Delta)\} \\
\cup {} & \{(\Delta; X; \sigma) \mid (\Delta; \tau; \sigma) \in \mathfrak{R} \text{ and } X \leq \tau \in \Delta\} \\
\cup {} & \{(\Delta; \mathsf{Class}[X_i \leq \delta_i \ ^{i\in 1..n}]\ I; \mathsf{Class}[X_i \leq \delta_i \ ^{i\in 1..n}]\ I') \mid \\
& \quad (\Delta, X_j \leq \delta_j \ ^{j\in 1..n}; I; I') \in \mathfrak{R}\} \\
\cup {} & \{(\Delta; I; \mu X.J) \mid (\Delta; I; J[X \leftarrow \mu X.J]) \in \mathfrak{R}\ \} \\
\cup {} & \{(\Delta; \mu X.I; J) \mid (\Delta; I[X \leftarrow \mu X.I]; J) \in \mathfrak{R} \text{ and } J \not\equiv \mu X.J'\ \} \\
\cup {} & \{(\Delta; \{m_i(\tau_{j_i} \ ^{j_i\in 1..n_i}) : \tau_i \ ^{i\in 1..n}\}; \{m_i(\tau'_{j_i} \ ^{j_i\in 1..n'_i}) : \tau'_i \ ^{i\in 1..n'}\}) \mid \\
& \quad n' \leq n \text{ and } n'_i = n_i \ \forall_{i\in 1..n'} \text{ and} \\
& \quad (\Delta; \tau_i; \tau'_i) \in \mathfrak{R} \ \forall_{i\in 1..n'} \text{ and } (\Delta; \tau'_{j_i}; \tau_{j_i}) \in \mathfrak{R} \ \forall_{j_i\in 1..n_i} \forall_{i\in 1..n'}\}.
\end{aligned}$$

We can verify that the mapping S is monotonic. Therefore, there exists its greatest fixed point $\nu S \in \mathcal{P}(\mathcal{J})$. We then define the subtyping relation as follows

Definition 3.2. $\Delta \vdash \tau \leq \sigma \triangleq (\Delta, \tau, \sigma) \in \nu S$.

The relation thus defined enjoys the basic properties of weakening, substitution of type variables, equivariance, narrowing and transitivity which are essential to prove the type safety of the language (Theorem 3.10) and the correctness of the subtyping algorithm (Theorem 4.10). In general, these kind of results are

proved by somewhat involved inductions on derivations; in our setting, due to the natural definition of subtyping as a greatest fixed point, we may handle them by quite standard coinductive proof techniques. We start by considering the weakening property in νS.

Proposition 3.3 (Weakening). *For all typing environments $\Delta, \Delta' \in \mathcal{D}$, and types $\tau, \sigma, \delta \in \mathcal{T}$, if $\Delta, \Delta' \vdash \tau \leq \sigma$, $X \notin Dom(\Delta, \Delta')$, and $\Delta \vdash \delta$ ok then $\Delta, X \leq \delta, \Delta' \vdash \tau \leq \sigma$.*

Proof. Consider the following set:

$$\mathfrak{W} \triangleq \{(\Delta, X \leq \delta, \Delta'; \tau; \sigma) \mid (\Delta, \Delta'; \tau; \sigma) \in \nu S \text{ and } X \notin Dom(\Delta, \Delta') \text{ and } FV(\delta) \subseteq Dom(\Delta)\}.$$

By case analysis in the definition of S we prove $\mathfrak{W}$ to be S-consistent, $\mathfrak{W} \subseteq S(\mathfrak{W})$, thus by the coinduction principle we have that $\mathfrak{W} \subseteq \nu S$. $\Diamond$

We use the same coinductive technique to prove that the substitution of type variables is sound, and that the subtyping relation is closed under name permutation.

Proposition 3.4 (Substitution of type variables). *For all $\Delta, \Delta' \in \mathcal{P}(\mathcal{J})$, and $\tau, \sigma, \delta, \delta' \in \mathcal{T}$, if $\Delta, X \leq \delta, \Delta' \vdash \tau \leq \sigma$ and $\Delta \vdash \delta' \leq \delta$ then we have $\Delta, \Delta'[X \leftarrow \delta'] \vdash \tau[X \leftarrow \delta'] \leq \sigma[X \leftarrow \delta']$.*

Proposition 3.5 (Equivariance). *For all $\Delta \in \mathcal{P}(\mathcal{J})$, $\tau, \sigma \in \mathcal{T}$, if $\Delta \vdash \tau \leq \sigma$ then $\Delta[X \leftrightarrow Y] \vdash \tau[X \leftrightarrow Y] \leq \sigma[X \leftrightarrow Y]$.*

We now prove that νS is transitive by considering a combined property where transitivity is expressed together with narrowing. We start by defining narrowing of typing environments, and then closure of νS under narrowing.

Definition 3.6. *For all $\Delta, \Delta' \in \mathcal{D}$, we have that Δ is narrower than Δ' with relation to $\mathfrak{R} \in \mathcal{P}(\mathcal{J})$, written $\Delta \sqsubseteq_{\mathfrak{R}} \Delta'$, where the relation $\sqsubseteq_{\mathfrak{R}}$ is inductively defined by letting $\emptyset \sqsubseteq_{\mathfrak{R}} \emptyset$ and*

$$\Gamma, X \leq \gamma \sqsubseteq_{\mathfrak{R}} \Gamma', X \leq \gamma' \quad \text{if } \Gamma \sqsubseteq_{\mathfrak{R}} \Gamma' \text{ and } (\Delta, \gamma, \gamma') \in \mathfrak{R}.$$

Definition 3.7. *For all $n \in \mathbb{N}$ we inductively define the sets $\mathfrak{N}^n$ as follows:*

$$\mathfrak{N}^0 \triangleq \nu S$$
$$\mathfrak{N}^n \triangleq \{(\Delta, \tau, \sigma) \mid (\Gamma, \tau, \sigma) \in \mathfrak{N}^{n-1} \text{ and } \Delta \sqsubseteq_{\mathfrak{N}^{n-1}} \Gamma\}.$$

We then define the transitive closure of νS, and prove that νS is closed under transitivity. Instead of a more direct definition, for technical convenience we present the transitivity relation based on chains of tuples with finite length.

Definition 3.8 (Extended Transitive Closure of νS).

$$\mathfrak{T} \triangleq \{(\Delta, \alpha_0, \alpha_n) \mid \exists_{n \in \mathbb{N}}. \exists_{\alpha_0 .. \alpha_n \in \mathcal{T}}. \forall_{i \in 0..n-1}. (\Delta, \alpha_i, \alpha_{i+1}) \in \mathfrak{N}^n \}$$

Notice that $\mathfrak{T}$ includes the transitive closure of νS ($n = 2$), and closure under narrowing ($n = 1$). We can then state and prove.

Proposition 3.9 (Transitivity). *If* $\Delta \vdash \tau \leq \sigma$ *and* $\Delta \vdash \sigma \leq \gamma$ *then* $\Delta \vdash \tau \leq \gamma$.

Proof. We show that $\mathfrak{T} \subseteq S(\mathfrak{T})$, using an inner induction on the number of tuples and by case analysis on the last rule used on the first tuple of a chain in $\mathfrak{T}$. We then conclude $\mathfrak{T} \subseteq \nu S$ by the coinduction principle. ◊

An interesting fact about our proof is that it exposes the loss of transitivity elimination pointed out in Ghelli's inductive system [11]. In the particular case of the variable transitivity, a tuple of $\mathfrak{T}$ supported by a chain of tuples in νS is supported in $S(\mathfrak{T})$ by a longer chain of tuples. Which nevertheless leads to the conclusion that the tuple is in the greatest fixed point of S.

3.3 Type Safety

We can now state and prove subject reduction for our class based programming language, that can then be used to show that well typed programs "don't go wrong" along usual lines.

Theorem 3.10 (Subject Reduction). *If* $\Delta \vdash e : \tau$ *and* $e \Downarrow v$ *then* $\Delta \vdash v : \tau'$ *where* $\Delta \vdash \tau' \leq \tau$.

Proof. By induction on the length of the typing derivations and by case analysis on the last rule used. We also use in several parts the fact that all valid subtyping judgments are supported in νS. ◊

As a result of this section we obtain declarative type and subtype systems whose implementability and decidability is far from being obvious (full detailed proofs for the results in this paper can be found in [20]). In the next section, we define and explain two simple algorithms that implement them.

4 Typing Algorithm

We define a type checking algorithm for our type system, thus proving that it is decidable. The algorithm is composed by two procedures: a typing algorithm that given an environment and a typable expression returns its the minimal type, and a subtyping algorithm, that is called by the typing procedure to verify the subtyping relations.

4.1 Typing Expressions

Typing of expressions is implemented by interpreting a set of rules bottom up: this defines a procedure that given a typing environment and a language expression returns a type. The new algorithmic rules are shown in Fig. 6, to these rules we must add (T-Var), (T-Let), (T-Fix), and (T-New), which are exactly as in Fig. 4. The resulting proof system is algorithmic because in all rules the resulting types are constructed either from the expression itself or from the types

$$\frac{\begin{pmatrix} c = \{m_i(x_{j_i} : \tau_{j_i}\ {}^{j_i \in 1..n_i}) : \tau_i = e_i\ {}^{i \in 1..n}\} \\ I = \{m_i(\tau_{j_i}\ {}^{j_i \in 1..n_i}) : \tau_i\ {}^{i \in 1..n}\} \end{pmatrix}}{\Delta, X_j \leq \delta_j\ {}^{j \in 1..m}, x_{j_i} : \tau_{j_i}\ {}^{j_i \in 1..n_i}, s : I \vdash_a e_i : \tau_i' \quad \Delta \vdash \tau_i' \leq \tau_i\ \ \forall_{i \in 1..n}}{\Delta \vdash_a \mathsf{class}[X_j \leq \delta_j\ {}^{j \in 1..m}](s)\ c : \mathsf{Class}[X_j \leq \delta_j\ {}^{j \in 1..m}]\ I} \quad \text{(A-Class)}$$

$$\frac{\Delta, x_{j_i} : \tau_{j_i}\ {}^{j_i \in 1..n_i} \vdash e_i : \tau_i' \qquad \Delta \vdash \tau_i' \leq \tau_i\ \ \forall_{i \in 1..n}}{\Delta \vdash \{\ m_i(x_{j_i} : \tau_{j_i}\ {}^{j_i \in 1..n_i}) = e_i\ {}^{i \in 1..n}\ \} : \{m_i(\tau_{j_i}\ {}^{j_i \in 1..n_i}) : \tau_i\ {}^{i \in 1..n}\}} \quad \text{(A-Object)}$$

$$\frac{\Delta \vdash_a e : \gamma \quad \Delta \vdash \gamma \Uparrow \gamma' \quad (\tau, \tau_i\ {}^{i \in 1..n}) = \mathit{lookup}(m, \gamma') \qquad \Delta \vdash_a e_i : \tau_i' \qquad \Delta \vdash \tau_i' \leq \tau_i\ \forall_{i \in 1..n}}{\Delta \vdash_a e.m(e_i\ {}^{i \in 1..n}) : \tau} \quad \text{(A-Call)}$$

$$\frac{X \leq \sigma \in \Delta \qquad \Delta \vdash \sigma \Uparrow \tau}{\Delta \vdash X \Uparrow \tau} \quad \text{(X-Var)} \qquad \frac{\tau \notin \mathit{Dom}(\Delta)}{\Delta \vdash \tau \Uparrow \tau} \quad \text{(X-Default)}$$

Fig. 6. Algorithmic typing rules

resulting from typing strictly smaller subexpressions. Notice that not every rule in Fig. 4 has a corresponding rule here; the (T-Sub) rule is not used in the algorithm as it depends on an unknown type that cannot be obtained neither from the expression nor from the types of the subexpressions, we replace it by subtyping verifications in rules (A-Class), (A-Object), and (A-Call). Moreover, we use an auxiliary function *lookup* to find a method in an interface and the judgment $\Delta \vdash \tau \Uparrow \sigma$, defined by the rules (X-Var) and (X-Default), to access the structure of type variables.

4.2 Subtyping Algorithm

We now present and prove correct our subtyping algorithm for deciding membership of a tuple $t \in \mathcal{J}$ in the greatest fixed point νS. The algorithm is defined by the recursive procedure shown in Fig. 7, and closely follows existing approaches for first-order equirecursive types [2, 3, 9]. Briefly, these algorithms progress by computing, given a pair of types to be checked for subsumption, a consistent set of pairs that includes it: by the coinduction principle, all the pairs in the set belong to the greatest fixed point. The consistent set is built by saturating the current approximation through backward rule application, and accumulating pairs of types, until a terminal case, corresponding to the application of an axiom, or an already visited pair is found.

We naturally extend those approaches building on the generating function in Definition 3.1, by defining an algorithm that manipulates judgments instead of pairs of types; this turns out to lead to a remarkably simple way of dealing with the binding information of the type variables. Notice that environments grow as a result of comparing polymorphic types, and, due to α-equivalence, the greatest fixed point νS is closed under renaming (Proposition 3.5). Moreover, in our setting, the number of tuples reachable from a given tuple are finite up to such renaming and pruning of useless variables (Lemma 4.5). Therefore, our

$$
\begin{array}{l}
\mathit{Subtyping}(A, (\Delta, \tau, \sigma)) = \\
\quad \mathsf{if}\ (\Delta, \tau, \sigma) \in^{\simeq} A\ \mathsf{then}\ A \\
\quad \mathsf{else\ let}\ A_0 = A \cup \{(\Delta, \tau, \sigma)\}\ \mathsf{in} \\
\quad \mathsf{if}\ \tau \equiv \sigma\ \mathsf{then}\ A \\
\quad \mathsf{else\ if}\ \sigma \equiv \mathsf{Top}\ \mathsf{then}\ A \\
\quad \mathsf{else\ if}\ \tau \equiv X\ \mathsf{then}\ \mathit{Subtyping}(A_0, (\Delta, \Delta(X), \sigma)) \\
\quad \mathsf{else\ if}\ \tau \equiv \mu X.\tau'\ \mathsf{then}\ \mathit{Subtyping}(A_0, (\Delta, \tau'[X \leftarrow \tau], \sigma)) \\
\quad \mathsf{else\ if}\ \tau \not\equiv \mu X.\tau'\ \mathsf{and}\ \sigma \equiv \mu X.\sigma'\ \mathsf{then}\ \mathit{Subtyping}(A_0, (\Delta, \tau, \sigma'[X \leftarrow \sigma])) \\
\quad \mathsf{else\ if}\ \tau \equiv \{m_i(\tau_{j_i}{}^{\,j_i \in 1..n_i}) : \tau_i{}^{\,i \in 1..n}\}\ \mathsf{and}\ \sigma \equiv \{m_i(\tau'_{j_i}{}^{\,j_i \in 1..n_i}) : \tau'_i{}^{\,i \in 1..n'}\}\ \mathsf{then} \\
\qquad A^{\ell}_{n_\ell}\ \mathsf{where}\ \ \forall_{i \in 1..n}\ (A^i_0 = \mathit{Subtyping}(A^{i-1}_{m_{i-1}}, (\Delta, \tau_i, \tau'_i))\ \mathsf{and} \\
\qquad\qquad\qquad\qquad \forall_{j \in 1..n_i}\ \ A^i_j = \mathit{Subtyping}(A^i_{j-1}, (\Delta, \tau'_{j_i}, \tau_{j_i}))) \\
\qquad \mathsf{with}\ A^0_{n_0} = A_0 \\
\quad \mathsf{else\ if}\ \tau \equiv \mathsf{Class}[X_i \le \delta_i{}^{\,i \in 1..n}]\ I\ \mathsf{and}\ \sigma \equiv \mathsf{Class}[X_i \le \delta_i{}^{\,i \in 1..n}]\ I'\ \mathsf{then} \\
\qquad \mathit{Subtyping}(A_0, (\Delta, X_i \le \delta_i{}^{\,i \in 1..n}; I; I')) \\
\quad \mathsf{else}\ \mathit{fail}
\end{array}
$$

Fig. 7. Subtyping algorithm

algorithm checks for membership of a tuple in the current approximation modulo a similarity relation on tuples that includes renaming, and allows us to detect cycles at the level of similarity equivalence classes, instead of expecting the exact tuple to reappear in the sequence of calls.

Definition 4.1 (Similarity). Similarity *is the binary relation* $\simeq$ *on* $\mathcal{J}$ *defined by:* $(\Delta, \tau, \sigma) \simeq (\Delta', \tau', \sigma')$*, if there are two typing environments* $\Gamma \subseteq \Delta$ *and* $\Gamma' \subseteq \Delta'$ *with* $\Gamma \vdash \tau\ \mathsf{ok}$*,* $\Gamma \vdash \sigma\ \mathsf{ok}$*, and* $\Gamma' \vdash \tau'\ \mathsf{ok}$*,* $\Gamma' \vdash \sigma'\ \mathsf{ok}$*, and a bijection* $\rho : \mathit{Dom}(\Gamma) \to \mathit{Dom}(\Gamma')$ *such that* $\rho(\Gamma) = \Gamma'$*,* $\rho(\tau) = \tau'$*,* $\rho(\sigma) = \sigma'$*.*

In the sequel we will use the following abbreviation $t \in^{\simeq} A \triangleq \exists u \in A. t \simeq u$, in particular in the first clause of the subtyping algorithm. Notice that similarity is decidable, it can be checked by matching the structure of the types, modulo bijective renaming of their free type variables, and recursively checking if the corresponding bounds are similar. All unused variables are discarded from the comparison as it goes through the structure of types and bounds of relevant type variables in the environments. For instance, the following two tuples are similar:

$$(X \le \{m(\tau){:}\tau\}, Y \le X; \mu Z.\{m(X){:}Z\}; X) \simeq (Z \le \{m(\tau){:}\tau\}; \mu X.\{m(Z){:}X\}; Z)$$

Notice the redundancy of Y, and that the tuples are similar with the permutation $[X \leftrightarrow Z]$.

An important fact is that the subtyping relation νS is closed under similarity.

Definition 4.2 (Closure under similarity). *For any* $R \in \mathcal{P}(\mathcal{J})$ *we define its closure under similarity, noted* R^**, as follows:*

$$R^* \triangleq \{t' \mid t' \in \mathcal{J}\ \textit{and}\ t \in R\ \textit{and}\ t' \simeq t\}.$$

Lemma 4.3. νS *is closed under similarity* ($\nu S^* = \nu S$)*.*

Proof. By substitution of type variables (Proposition 3.4), equivariance (Proposition 3.5) and then by weakening (Proposition 3.3). ◊

We now prove the correctness and decidability of the subtyping algorithm. We first show that the algorithm terminates on all inputs. This is done by showing that the search space of the algorithm is finite in some sense. To characterize such search space we introduce the following reachability relation:

Definition 4.4 (Reachability). *Reachability is the binary relation on $\mathcal{J}$, noted $t \gg t'$, inductively defined as follows:*

1. $(\Delta, \tau, \sigma) \gg (\Delta, \tau, \sigma)$
2. *if* $(\Delta, \tau, \sigma) \gg (\Delta', X, \sigma')$ *then* $(\Delta, \tau, \sigma) \gg (\Delta', \Delta'(X), \sigma')$
3. *if* $(\Delta, \tau, \sigma) \gg (\Delta', \mathsf{Class}[X_i \leq \delta_i \ ^{i\in 1..n}]\ I; \mathsf{Class}[X_i \leq \delta_i \ ^{i\in 1..n}]\ I')$
   then $(\Delta, \tau, \sigma) \gg (\Delta', X_j \leq \delta_j \ ^{j\in 1..n}; I; I')$
4. *if* $(\Delta, \tau, \sigma) \gg (\Delta'; I; \mu X.J)$ *then* $(\Delta, \tau, \sigma) \gg (\Delta'; I; J[X \leftarrow \mu X.J])$
5. *if* $(\Delta, \tau, \sigma) \gg (\Delta'; \mu X.I; J)$ *then* $(\Delta, \tau, \sigma) \gg (\Delta'; I[X \leftarrow \mu X.I]; J)$
6. *if* $(\Delta, \tau, \sigma) \gg (\Delta'; \{m_i(\tau_{j_i} \ ^{j_i\in 1..n_i}) : \tau_i \ ^{i\in 1..n}\}; \{m_i(\tau'_{j_i} \ ^{j_i\in 1..n'_i}) : \tau'_i \ ^{i\in 1..n'}\})$
   then $(\Delta, \tau, \sigma) \gg (\Delta'; \tau_i; \tau'_i)\ \forall_{i\in 1..n'}$
   and $(\Delta, \tau, \sigma) \gg (\Delta'; \tau'_{j_i}; \tau_{j_i})\ \forall_{j_i\in 1..n_i} \forall_{i\in 1..n'}$

We let $Reach(t) \triangleq \{t' \mid t \gg t'\}$. We have the following

Lemma 4.5. *$Reach(t)_{/\simeq}$ is finite.*

Proof. We prove by induction on the reachability relation that all types occurring in reachable tuples are subexpressions of the initial tuple and that the number of relevant type variables in those tuples, both in the environments and in the type expressions, decreases with relation to the initial tuple. We then prove by contradiction that these two results support the fact that the set of reachable tuples, $Reach(t)$, is finite modulo similarity. ◊

It is important to remark that the finite reachability property of Lemma 4.5 holds both for Kernel-Fun and $F^{\top}_{\leq}$ style subtyping, although in the first case the proof is slightly more involved (see [20]). The proof also enlightens why the same result cannot be extended to $F_{\leq}$.

Theorem 4.6. *For all $A \in \mathcal{P}(\mathcal{J})$, and $t \in \mathcal{J}$, Subtyping(A, t) terminates.*

Proof. By induction using a measure that represents the number of non-visited equivalence classes of reachable tuples of t. Since the algorithm always increases the visited tuples with a tuple reachable from t, the measure decreases in all cases. Hence, the algorithm terminates. ◊

To prove that our algorithm is sound and complete, it is technically convenient to follow the approach of [9] and introduce a function *gfp* that allows us to characterize νS in a form both suitable for the correctness proofs and for establishing the correspondence between the algorithm and the extensional definition of the subtyping relation. Moreover, unlike the analogous notion in [9], instead of accumulating tuples, our *gfp* function works with $\simeq$ equivalence classes.

Definition 4.7. *Let gfp be the partial function* $\mathcal{P}(\mathcal{J}) \times \mathcal{J} \to \mathcal{P}(\mathcal{J})$ *defined by:*

$$
\begin{array}{l}
gfp(A,t) = \mathsf{if}\ \{t\}^* \subseteq A\ \mathsf{then}\ A \\
\qquad\qquad \mathsf{else\ if}\ support(t)\ \textit{is undefined}\ \mathsf{then}\ \textit{undefined} \\
\qquad\qquad \mathsf{else\ let}\ \{t_1,\ldots,t_n\} = support(t)\ \mathsf{in} \\
\qquad\qquad\ \ \mathsf{let}\ A_0 = A \cup \{t\}^*\ \mathsf{in} \\
\qquad\qquad\ \ \mathsf{let}\ A_1 = gfp(A_0,t_1)\ \mathsf{in} \\
\qquad\qquad\ \ \ldots \\
\qquad\qquad\ \ \mathsf{let}\ A_n = gfp(A_{n-1},t_n)\ \mathsf{in}\ A_n.
\end{array}
$$

We prove next that *gfp* indeed characterizes the subtyping relation.

Lemma 4.8 (Correctness of *gfp*). *For all* $t \in \mathcal{J}$*, and* $A \in \mathcal{P}(\mathcal{J})$*,*

1. *if* $gfp(\emptyset,t) = A$ *then* $t \in \nu S$*.*
2. *if* $gfp(\emptyset,t)$ *is undefined then* $t \notin \nu S$*.*

Proof. 1. By induction in the definition of *gfp* to prove the following more general property: for all $A, A' \in \mathcal{P}(\mathcal{J})$, and $t \in \mathcal{J}$, if $A^* \subseteq A$ and $gfp(A,t) = A'$ then $A \subseteq A'$, $A'^* \subseteq A'$, $\{t\}^* \subseteq A'$, and $A' \subseteq S(A') \cup A$. We then conclude by considering $A = \emptyset$. 2. Also by induction on the definition of *gfp*. ◊

Knowing that *gfp* correctly checks if any tuple belongs to the subtyping relation νS, we prove that *gfp* and the subtyping algorithm *Subtyping* are equivalent. Since the algorithm terminates on all inputs (Theorem 4.6), we can show that it is sound and complete.

Lemma 4.9 (Correctness of *Subtyping*). *For all* $t \in \mathcal{J}$*,* $A, A' \in \mathcal{P}(\mathcal{J})$*,*

1. $Subtyping(A,t) = A'$ *iff* $gfp(A^*,t) = A'^*$*.*
2. $Subtyping(A,t) = fail$ *iff* $gfp(A^*,t)$ *is undefined.*

Proof. By induction on the recursive calls of *Subtyping*. ◊

Theorem 4.10 (Correctness of *Subtyping*). *For all* $\Delta \in \mathcal{D}$*, and* $\tau, \sigma \in \mathcal{T}$*,*

$$Subtyping(\emptyset, (\Delta, \tau, \sigma)) = A \text{ iff } \Delta \vdash \tau \leq \sigma$$

Proof. It follows directly from Lemmas 4.8 and 4.9. ◊

To summarize, we conclude that the type system defined in section 3 is decidable and that our typing and subtyping algorithms are sound and complete.

5 Composition of Classes

We have presented a language that treats classes as first class values but that lacks class composition operations, so that no new class values can be actually created at runtime. In this section, we discuss an interesting and possible extension of this object language so to include class manipulation mechanisms,

$$\left(\begin{array}{l}\pi = [X'_i \leftarrow \gamma_i \ ^{i\in 1..n'}] \qquad \pi' = [X''_i \leftarrow \gamma'_i \ ^{i\in 1..n''}] \\ I = \{m_i(\vec{\sigma}\ \pi) : \tau_i\pi \ ^{i\in\mathcal{I}-\mathcal{J}},\ m_i(\vec{\sigma'}\ \pi') : \tau'_i\pi' \ ^{i\in\mathcal{J}}\}\ )\end{array}\right)$$

$$\Delta \vdash e_1 : \mathsf{Class}[X'_i \leq \delta'_i \ ^{i\in 1..n'}]\ \{m_i(\vec{\sigma_i}) : \tau_i \ ^{i\in\mathcal{I}}\} \quad \Delta \vdash \gamma_i \leq \delta'_i \ \ \forall_{i\in 1..n'}$$

$$\Delta \vdash e_2 : \mathsf{Class}[X''_i \leq \delta''_i \ ^{i\in 1..n''}]\ \{m_i(\vec{\sigma'_i}) : \tau'_i \ ^{i\in\mathcal{J}}\} \quad \Delta \vdash \gamma'_i \leq \delta''_i \ \ \forall_{i\in 1..n''}$$

$$\Delta \vdash \mathsf{mix}[X_i \leq \delta_i \ ^{i\in 1..n}](e_1[\gamma_i \ ^{i\in 1..n'}] \triangleleft e_2[\gamma'_i \ ^{i\in 1..n''}]) : \mathsf{Class}[X_i \leq \delta_i \ ^{i\in 1..n}]\ I$$

$$\left(\begin{array}{l}\pi = [s \leftarrow s_{(m_i \leftrightarrow m'_i \ ^{i\in\mathcal{I}\cap\mathcal{J}})}]\ (m'_i \text{ fresh}) \quad \pi' = [X'_i \leftarrow \gamma_i \ ^{i\in 1..n'}][X''_i \leftarrow \gamma'_i \ ^{i\in 1..n''}] \\ v = \{m_i(\vec{\sigma_i}) : \tau_i = e_i\pi \ ^{i\in\mathcal{I}-\mathcal{J}},\ m_i(\vec{\sigma'_i}) : \tau'_i = e'_i \ ^{i\in\mathcal{J}},\ m'_i(\vec{\sigma_i}) : \tau_i = e_i\pi \ ^{i\in\mathcal{I}\cap\mathcal{J}}\}\end{array}\right)$$

$$e_1 \Downarrow \mathsf{class}[X'_i \leq \delta'_i \ ^{i\in 1..n'}](s)\ \{m_i(\vec{\sigma_i}) : \tau_i = e_i \ ^{i\in\mathcal{I}}\}$$

$$e_2 \Downarrow \mathsf{class}[X''_i \leq \delta''_i \ ^{i\in 1..n''}](s)\ \{m_i(\vec{\sigma'_i}) : \tau'_i = e'_i \ ^{i\in\mathcal{J}}\}$$

$$\mathsf{mix}[X_i \leq \delta_i \ ^{i\in 1..n}](e_1[\gamma_i \ ^{i\in 1..n'}] \triangleleft e_2[\gamma'_i \ ^{i\in 1..n''}]) \Downarrow \mathsf{class}[X_i \leq \delta_i \ ^{i\in 1..n}](s)\ v\pi'$$

$$\frac{e \Downarrow v}{e_{(m\leftrightarrow m')} \Downarrow v(m \leftrightarrow m')} \qquad \frac{\Delta \vdash e : \tau}{\Delta \vdash e_{(m\leftrightarrow m')} : \tau(m \leftrightarrow m')}$$

Fig. 8. Typing and evaluation for mix

similar to inheritance or mixin application. As an alternative, we propose a general mechanism of class composition which combines two classes without any of them having to be developed with extension in mind. The composition mechanism is expressed as follows:

$$\mathsf{mix}[X_i \leq \delta_i \ ^{i\in 1..n}](e_1[\gamma_i \ ^{i\in 1..n'}] \triangleleft e_2[\gamma'_i \ ^{i\in 1..n''}])$$

It takes two class values, e_1 and e_2, and produces a new class value, parameterized by a fresh set of type parameters, containing the methods in e_1 and e_2 and where name clashes are resolved in favor of e_2. This is apparent in the rules in Fig. 8 which must be added to the type and evaluation systems to extend the initial language. Notice that for a mix to be well typed e_1 and e_2 must denote class values, and that the arguments $\gamma_i \ ^{i\in 1..n'}$ and $\gamma'_i \ ^{i\in 1..n''}$ must be compatible with the bounds of each class value.

It is well known that extending a class by simply replacing the methods of one class with methods of another easily generates type inconsistencies (*e.g.* see [1, 6]). To avoid this, we maintain the local coherence of the subsumed class, which in this case is e_1, by means of an explicit permutation of method names, $e_{(m\leftrightarrow m')}$. This corresponds to the run-time permutation of method names between m and m' and $\tau(m \leftrightarrow m')$ has the same meaning but for type expressions. The typing and evaluation rules for this expression are also depicted in 8.

So, whenever s, now replaced by $s_{(m_i\leftrightarrow m'_i \ ^{i\in\mathcal{I}\cap\mathcal{J}})}$, is evaluated in the body of a method of e_1, the methods in s are switched back to the subsumed methods of e_1 that were hidden under a different name. The evaluation of such methods and possibly of other methods using the self reference passed by a method of e_1 is type preserving, and thus the core language extended with mix can be shown to enjoy a subject reduction property [20].

6 Related Work and Concluding Remarks

It is known that the type system of $F_{\leq}$ is not decidable [12, 16] and that its extension with recursive types is non-conservative [11]. Simpler versions were proposed but even so, not free from problems: system $F_{\leq}^{\top}$, proposed by Castagna and Pierce, is decidable revealed to lack the minimal typing property [4] and the approach to subtyping in Kernel Fun extended with recursive types by Colazzo and Ghelli [5] results in a fairly complex algorithm; it uses a labeling mechanism to identify cycles in derivations and stop the unfolding process. The authors show that the use of renaming of type variables results in an divergent algorithm and that this labeling technique is sound. However, the algorithm only stops unfolding a given pair of types the third time it occurs. The reasons for this fact are far from intuitive. On the contrary, our approach and algorithm are a natural extension of well known techniques for first-order types. Alan Jeffrey defines in [14] a notion of simulation for higher order types using a symbolic labeled transition systems, and uses it to define a (non decidable) subtyping relation in $\mu F_{\leq}$, there is then some connection between his approach and the general coinductive techniques we have presented here. An efficient algorithm to unify two recursive second-order types was proposed by Gauthier and Pottier in [10]. It relies on an encoding of second-order type expressions into first-order trees, and on the application of standard first-order unification algorithms for infinite trees. We have no perspective on how this may be adapted to the subtyping problem.

On the other hand, we present a subtyping algorithm for second-order systems with equirecursive types which is an uniform extension of existing work on first-order equirecursive types by Amadio and Cardelli [2], Brandt and Henglein [3]. In particular, we build on the coinductive presentations of first-order type systems with recursive types of Gapeyev, Levin, and Pierce [9, 17]. Our treatment of reachability modulo a similarity relation that includes equivariance (Lemma 3.5) is inspired on notions by Gabbay and Pitts [8]. Our definition and correctness proof is, from our point of view, much simpler than the ones of the algorithms referred above. Our proofs are modular, in the sense that they can be applied to any polymorphic type system with recursive types that satisfies a certain finite reachability condition up to a notion of similarity that includes equivariance. In particular, they suggest an interesting decidable fragment of $F_{\leq}$, defined by restricting the subtype rule for $\forall X \leq \tau.\sigma$ types to just compare bounds with the same free type variables, we leave this topic for future work.

Given these results, we develop a class based language and discuss a possible extension of it that allows combination of classes with a mixin like construct, while avoiding the unsoundness problems of subsumption and class extension often found in object calculi.

We would like to thank Dario Colazzo and the reviewers for their comments on a preliminary version of this work. This work is partially supported by FCT/MCES.

References

1. Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.
2. Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.
3. Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In Roger Hindley, editor, *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2–4, 1997*, volume 1210, pages 63–81. Springer-Verlag, 1997.
4. Giuseppe Castagna and Benjamin Pierce. Corrigendum: Decidable bounded quantification. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*. ACM, January 1995.
5. Dario Colazzo and Giorgio Ghelli. Subtyping recursive types in Kernel Fun. In *14th Symp. on Logic in Computer Science (LICS'99)*, pages 137–146, 1999.
6. William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 497–518. The MIT Press, Cambridge, MA, 1994.
7. Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. In *Theoretical aspects of object-oriented programming: types, semantics, and language design*, pages 247–292. MIT Press, 1994.
8. M. Gabbay and A. Pitts. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing*, 13(3–5):341–363, 2002.
9. Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed. In *Proc. of the Intl. Conference on Functional Programming (ICFP)*, 2000.
10. Nadji Gauthier and François Pottier. Numbering matters: First-order canonical forms for second-order recursive types. In *Proc. of the 2004 ACM SIGPLAN Intl. Conference on Functional Programming (ICFP'04)*, pages 150–161, 2004.
11. Giorgio Ghelli. Recursive types are not conservative over $F_{\leq}$. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculus and Applications*, volume 664 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1993.
12. Giorgio Ghelli. Divergence of $F_{\leq}$ type checking. *Theoretical Computer Science*, 139(1-2):131–162, 1995.
13. Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proc. of the 1999 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.
14. Alan Jeffrey. A symbolic labelled transition system for coinductive subtyping of $F_{\mu\leq}$. In *16th Annual IEEE Symposium on Logic in Computer Science*, June 2001.
15. Betti Venneri Lorenzo Bettini, Viviana Bono. Subtyping mobile classes and mixins. In *FOOL 10*, 2003.
16. Benjamin C. Pierce. Bounded quantification is undecidable. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 427–459. MIT Press, 1994.
17. Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
18. João Costa Seco and Luís Caires. A basic model of typed components. In *Proc. of the European Conf. on Object-Oriented Programming (ECOOP)*, 2000.
19. João Costa Seco and Luís Caires. The parametric component calculus. Technical Report UNL-DI-7-2002, FCT-UNL, 2002.
20. João Costa Seco and Luís Caires. Subtyping first class polymorphic components. Technical Report UNL-DI-1-2005, FCT-UNL, 2004.

Complexity of Subtype Satisfiability over Posets

Joachim Niehren[1], Tim Priesnitz[2], and Zhendong Su[3]

[1] INRIA Futurs, Lille, France
[2] Programming Systems Lab, Saarland University, Saarbrücken, Germany
[3] Department of Computer Science, University of California, Davis, CA 95616 USA

Abstract. Subtype satisfiability is an important problem for designing advanced subtype systems and subtype-based program analysis algorithms. The problem is well understood if the atomic types form a lattice. However, little is known about subtype satisfiability over posets. In this paper, we investigate algorithms for and the complexity of subtype satisfiability over general posets. We present a uniform treatment of different flavors of subtyping: simple versus recursive types and structural versus non-structural subtype orders. Our results are established through a new connection of subtype constraints and modal logic. As a consequence, we settle a problem left open by Tiuryn and Wand in 1993.

1 Introduction

Many programming languages have some form of subtyping. The most common use is in the sub-classing mechanisms in object-oriented languages. Also common is the notion of "coercion" [17], for example automatic conversion from integers to floating point numbers.

Type checking and type inference for subtyping systems have been extensively studied since the original results of Mitchell [18]. The main motivations for investigating these systems today are more advanced designs for typed languages and program analysis algorithms based on subtyping.

Subtyping systems invariably involve *subtype constraints*, inequalities of the form $t_1 \leq t_2$, to capture that the type t_1 is a *subtype* of t_2. For example, the constraint $int \leq real$ means that at any place a floating point number is expected, an integer can be used instead. Besides of type constants, subtype constraints may contain type variables and type constructors, such as the constraint $int \times x \leq x \times real$ that is equivalent to $int \leq x \leq real$.

Type variables are typically interpreted as trees built from type constants and type constructors. The trees can be infinite if recursive types are allowed. There are two choices for the subtype relation. In a system with *structural subtyping* only types with the same shape are related. In a system with *non-structural subtyping*, there is a "least" type $\bot$ and a "largest" type $\top$ that can be related to types of arbitrary shape.

Three logical problems for subtype constraints are investigated in the literature: satisfiability [1,5,9,13,14,18,23,26,32,33], entailment [7,11,12,19,20,24,25,28,34], and first-order validity [16,31]. In this paper, we close a number of problems on satisfiability.

If the type constants form a lattice then subtype satisfiability is well understood [14, 18, 23]. For general partially-ordered sets (posets), however, there exist only

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 357–373, 2005.

Table 1. Summary of complexity results on subtype satisfiability over posets

	structural	non-structural
finite types	*PSPACE* (Frey, 1997 [8]) *PSPACE*-hard (Tiuryn, 1992 [32])	*PSPACE*-complete ($\star$)
recursive types	*DEXPTIME* (Tiuryn and Wand, 1993 [33]) *DEXPTIME*-hard ($\star$)	*DEXPTIME*-complete ($\star$)

partial answers. Tiuryn and Wand show that recursive structural satisfiability is in *DEXPTIME* [33]. Tiuryn shows that finite structural satisfiability is *PSPACE*-hard [32], and subsequently Frey shows that it is in *PSPACE* and thus *PSPACE*-complete [8]. Decidability and complexity of non-structural subtype satisfiability are open, for both finite and recursive types.

We summarize here the main contributions of this paper. We close the open questions on subtype satisfiability over posets. We consider all combinations of finite versus recursive types, and structural versus non-structural orders.

We base our results on a new approach, connecting subtype constraints and modal logic. We introduce *uniform subtype constraints* and show that their satisfiability problem is polynomial time equivalent to that of a dialect of *propositional dynamic logic* [2,4,6], which is subsumed by the monadic second-order logic SnS of the complete infinite n-ary tree [27]. With this connection, we completely characterize the exact complexity of subtype satisfiability over posets in all cases.

Table 1 summarizes complexity results regarding subtype satisfiability over posets. New results of this paper are marked with "$\star$". In particular, we show in this paper, that recursive structural satisfiability is *DEXPTIME*-hard, finite non-structural satisfiability is *PSPACE*-complete, and recursive non-structural satisfiability is *DEXPTIME*-complete. This settles a longstanding problem left open by Tiuryn and Wand in 1993 [33].

Due to space limitations, we omit some of the proofs. Interested readers can refer to the full paper [21] for more details.

2 Subtyping

In this section, we formally define satisfiability problems of subtype constraints.

2.1 Types as Trees

Types can be viewed as trees over some ranked alphabet Σ, the *signature* of the given type language. A signature consists of a finite set of function symbols (a.k.a. *type constructors and constants*). Each function symbol f has an associated $arity(f) \geq 0$, indicating the number of arguments that f expects. Symbols with arity zero are *type constants*. The signature fixes for all type constructors f and all positions $1 \leq i \leq arity(f)$ a *polarity* $pol(f,i) \in \{1,-1\}$. We call a position i of symbol f *covariant* if $pol(f,i) = 1$ and *contravariant* otherwise.

We identify *nodes* π of trees with relative addresses from the root of the tree, *i.e.*, with words in $(\mathbb{N} - \{0\})^*$. A word πi addresses the i-th child of node π, and $\pi\pi'$ the π'

descendant of π. The root is represented by the empty word ε. We define a *tree* τ over Σ as a partial function: $\tau : (\mathbb{N} - \{0\})^* \to \Sigma$. Tree domains $dom(\tau)$ are prefixed closed, non-empty, and arity consistent, *i.e.*: $\forall \pi \in dom(\tau) \forall i \in \mathbb{N} : \pi i \in dom(\tau) \leftrightarrow 1 \leq i \leq arity(\tau(\pi))$. A tree τ is *finite* if $dom(\tau)$ is a finite set, and *infinite* otherwise. We write $tree_\Sigma$ for the set of possibly infinite trees over Σ.

Given a function symbol f with $n = arity(f)$ and trees $\tau_1, \ldots, \tau_n \in tree_\Sigma$ we define $f(\tau_1, \ldots, \tau_n)$ as the unique tree τ with $f(\tau_1, \ldots, \tau_n)(\varepsilon) = f$ and $f(\tau_1, \ldots, \tau_n)(i\pi) = \tau_i(\pi)$. We define the *polarities* of nodes in trees as follows:

$$\begin{array}{ll} pol_\tau(\varepsilon) & =_{\mathrm{df}} 1 \\ pol_{f(\tau_1,\ldots,\tau_n)}(i\pi) & =_{\mathrm{df}} pol(f,i) * pol_{\tau_i}(\pi) \end{array}$$

For partial orders $\leq$, let $\leq^1$ denote the order $\leq$ itself and $\leq^{-1}$ the reversed relation, $\geq$.

Subtype orders $\leq$ are partial orders on trees over some signature Σ. Two subtype orders arise naturally, *structural subtyping* and *non-structural* subtyping.

2.2 Structural Subtyping

We investigate structural subtyping with signatures Σ that provide the standard type constructors $\times$ and $\to$ and a poset $(B, \leq_B)$ of type constants, i.e., $\Sigma = B \cup \{\times, \to\}$. The product type constructor $\times$ is a binary function symbol that is covariant in both positions ($pol(\times, 1) = pol(\times, 2) = 1$), while the function type constructor $\to$ is contravariant in its first and covariant in its second argument ($pol(\to, 1) = -1$ and $pol(\to, 2) = 1$).

Structural subtype orders $\leq$ are partial orders on trees over structural signatures Σ. They are obtained by lifting the ordering on constants $(B, \leq_B)$ in Σ to trees. More formally, $\leq$ is the smallest binary relation $\leq$ on $tree_\Sigma$ such that for all $b, b' \in B$ and types τ_1, τ_2, τ_1', τ_2' in $tree_\Sigma$:

- $b \leq b'$ iff $b \leq_B b'$;
- $\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'$ iff $\tau_1 \leq \tau_1'$ and $\tau_2 \leq \tau_2'$;
- $\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$ iff $\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$.

Notice that $\times$ is monotonic in both of its arguments while $\to$ is anti-monotonic in its first argument and monotonic in its second. For more general signatures, monotonic arguments are specified by covariant positions of function symbols, and anti-monotonic arguments by contravariant positions.

For structural subtyping, two types are related only if they have exactly the same shape, *i.e.*, tree domain. Notice that structural subtype orders are indeed partial orders. We do not restrict ourselves to lattices $(B, \leq_B)$ in contrast to most previous work.

2.3 Non-structural Subtyping

In the *non-structural subtype order*, two distinguished constants are added to structural type languages, a *smallest type* $\bot$ and a *largest type* $\top$. The ordering is parametrized by a poset $(B, \leq_B)$ and has the signature: $\Sigma = B \cup \{\times, \to\} \cup \{\bot, \top\}$. For the non-structural subtype order, besides the three structural rules earlier, there is an additional requirement: $\bot \leq \tau \leq \top$ for any $\tau \in tree_\Sigma$.

2.4 Uniform Subtyping

We introduce *uniform subtyping* as an intermediate ordering for two reasons: (i) to capture both structural and non-structural subtyping effects and (ii) to bridge from uniform subtype constraints to modal logic.

We call a signature Σ *uniform* if all symbols in Σ have the same non-zero arity and the same polarities. All trees over Σ are complete infinite n-ary trees, where n is the arity common to all function symbols in Σ. Hence, all trees have the same shape. Furthermore, the polarities of nodes $\pi \in \{1,\ldots,n\}^*$ in trees τ over uniform signatures do not depend on τ. We therefore write $pol(\pi)$ instead of $pol_\tau(\pi)$.

The signatures $\{\times\}$ and $\{\to\}$, for instance, are both uniform, while $\{\times,\to\}$ or $\{\bot,\top,\times\}$ are not. The idea to model the non-structural signature $\{\bot,\top,\times\}$ uniformly is to raise the arities of $\bot$ and $\top$ to 2 and to order them by $\bot \leq_\Sigma \times \leq_\Sigma \top$.

A *uniform subtype order* $\leq$ is defined over a partially-ordered uniform signature $(\Sigma,\leq_\Sigma)$. It satisfies for all trees $\tau_1,\tau_2 \in tree_\Sigma$:

$$\tau_1 \leq \tau_2 \quad \text{iff} \quad \forall \pi \in \{1,\ldots,n\}^* :\ \tau_1(\pi) \leq_\Sigma^{pol(\pi)} \tau_2(\pi)$$

where n is the arity of the function symbols in Σ. For simplicity, we will often write $\leq_\Sigma^\pi$ instead of $\leq_\Sigma^{pol(\pi)}$.

2.5 Subtype Constraints and Satisfiability

In a subtype system, *type variables* are used to denote unknown types. We assume that there are a denumerable set of type variables $x,y,z \in V$. We assume w.l.o.g. that subtype constraints are *flat*, and subtype constraints φ over a signature Σ satisfy:

$$\varphi \ ::=\ x{=}f(x_1,\ldots,x_n) \mid x{\leq}y \mid \varphi \wedge \varphi$$

where n is the arity of $f \in \Sigma$. We call atomic constraints $x{=}f(x_1,\ldots,x_n)$ and $x{\leq}y$ the *literals*. The type variables in a constraint φ are called the *free variables* of φ, denoted by $V(\varphi)$.

We always consider two possible interpretations of subtype constraints, over possibly infinite tree over Σ, and over finite trees over Σ respectively. A variable assignment α is a function mapping type variables in V to trees over Σ. A constraint φ is *satisfiable* over Σ if there is a variable assignment α such that $\alpha(\varphi)$ holds in Σ.

We distinguish three subtype satisfiability problems, each of which has two variants depending on interpretation over finite or possibly infinite trees.

Structural subtype satisfiability is the problem to decide whether a structural subtype constraint is satisfiable. The arguments of this problem are a posets $(B,\leq_B)$ and a constraint φ over the signature $B \cup \{\times,\to\}$.

Non-structural subtype satisfiability is the problem to decide whether a non-structural subtype constraint is satisfiable. The arguments are a poset $(B,\leq_B)$ and a constraint φ over signature $B \cup \{\times,\to\} \cup \{\bot,\top\}$.

Uniform subtype satisfiability is the problem to decide whether a uniform subtype constraint is satisfiable. The arguments are a partially-ordered uniform signature $(\Sigma,\leq_\Sigma)$ and a subtype constraint φ over this signature.

$R ::= i \mid R \cup R' \mid RR' \mid R^*$ where $1 \leq i \leq n$
$A ::= p \mid \neg A \mid A \wedge A' \mid [R]A$

Fig. 1. Syntax of PDL_n

3 Propositional Dynamic Logic over Trees

Propositional dynamic logic (*PDL*) is a modal logic that extends Boolean logic to directed graphs of possible worlds. The same proposition may hold in some node of the graph and be wrong in others. Nodes are connected by labeled edges, that can be talked about modal operators.

In this paper, we consider the modal logic PDL_n, the *PDL* language for the complete infinite n-ary tree. PDL_n is naturally subsumed by the monadic second-order logic SnS of the complete n-ary tree [27].

3.1 Other PDL Dialects

Propositional dynamic logic (*PDL*) over directed edge-labeled graphs goes back to Fischer and Ladner [6], who restricted Pratt's dynamic logic to the propositional fragment. It is well known that *PDL* has the *tree property*: every satisfiable *PDL* formula can be satisfied in a rooted edge-labeled tree. Deterministic *PDL* [2, 10, 35] restricts the model class to graphs whose edge labels are functional in that they determine successor nodes. Deterministic *PDL* with edge labels $\{1, \ldots, n\}$ is the closest relative to our language PDL_n, due to the tree property.

Besides of PDL_n, a large variety of PDL dialects with tree models were proposed in the literature. These differ in the classes of tree models, the permitted modal operators, and the logical connectives. Three different dialects of PDL over finite, binary, or n-ary trees were proposed in [4, 15, 22], see [3] for a comparison. PDL over finite unranked ordered trees were proposed for computational linguistics applications [4] and found recent interest for querying XML documents.

3.2 PDL_n and Its Fragments

For every $n \geq 1$ we define a logic PDL_n as the PDL logic, for describing the complete infinite n-ary tree.

The syntax of PDL_n expressions[1] A is given in Figure 1. Starting from some infinite set P of propositional variables $p \in P$, it extends the Boolean logic over these variables by universal modalities $[R]A$, where R is a regular expression over the alphabet $\{1, \ldots, n\}$.

We frequently use the modality $[*]$ as an abbreviation of $[\{1, \ldots, n\}^*]$, and sometimes $[+]$ as a shorthand for $[\{1, \ldots, n\}^+]$. We freely use definable logical connective for implication $\rightarrow$, equivalence $\leftrightarrow$, disjunction $\vee$, exclusive disjunction $\dot{\vee}$, and the Boolean

[1] We could allow for test $?A$ in regular expressions, which frequently occur in PDL dialects but we will not need them.

Table 2. Semantics of PDL_n

$M, \pi \models p$	if $M(p, \pi) = 1$
$M, \pi \models A_1 \wedge A_2$	if $M, \pi \models A_1$ and $M, \pi \models A_2$
$M, \pi \models \neg A$	if not $M, \pi \models A$
$M, \pi \models [R]A$	if for all $\pi' \in L(R)$: $M, \pi\pi' \models A$

$B ::= p_1 \wedge p_2 \mid \neg p \mid [i]p$ where $1 \leq i \leq n$
$C ::= p \mid [*](p \leftrightarrow B) \mid C_1 \wedge C_2$

Fig. 2. Syntax of flat core PDL_n

constants $true$ and $false$. Furthermore, we can define existential modalities $\langle R\rangle A$ by $\neg[R]\neg A$.

We interpret formulas of PDL_n over the complete infinite n-ary trees. Tree nodes are labeled by the set of propositions that are valid there. Formally, a model M of a formula in PDL_n assigns Boolean values $0, 1$ to propositional variables in every node in $\{1, \ldots, n\}^*$, *i.e.*, $M : P \times \{1, \ldots, n\}^* \to \{0, 1\}$. Table 2 defines when a formula A holds in some node π of some model M, in formulas: $M, \pi \models A$. A formula $[R]A$ is valid for some node π of a tree M if A holds in all R descendants of π in M, *i.e.*, in all nodes $\pi\pi'$ where π' belongs to the language $L(R)$ of R.

Let us recall some logical notations. A formula A is *valid in a model* M if it holds in the root of M: $M \models A$ iff $M, \varepsilon \models A$. A formula A is *satisfiable* if it is valid in some model; it is *valid* it is valid in all models: $\models A$ iff $\forall M.\, M \models A$. Two formulas A, A' are equivalent if $A \leftrightarrow A'$ is valid: $A \models\!\models A'$ iff $\models A \leftrightarrow A'$. For instance, $\langle i\rangle A \models\!\models [i]A$ holds for all $1 \leq i \leq n$ and all A, since all nodes of the n-ary tree have unique i successors.

Note that PDL_n respects the substitution property: whenever $A_1 \models\!\models A_2$ then $A[A_1/A_2] \models\!\models A$. To see this note that if $A_1 \models\!\models A_2$ then the equivalence $A \leftrightarrow A'$ is valid not only at the root of all models but also at all other nodes of all models. This is because all subtrees of complete n-ary trees are again complete n-ary trees.

Theorem 1. *Satisfiability of PDL_n formulas is in DEXPTIME.*

A PDL_n formula is satisfiable iff it can be satisfied by a deterministic rooted graph with edge labels in $\{1, \ldots, n\}$. The theorem thus follows from the *DEXPTIME* upper bound for deterministic *PDL* [2, 10], which follows from the analogous result for *PDL*.

3.3 Flat Core PDL_n

We next investigate lower complexity bounds for PDL_n. It is known from Vardi and Wolper [35] that satisfiability of deterministic *PDL* is *DEXPTIME*-complete. This result clearly carries over to PDL_n.

An analysis of Spaan's proofs [30] reveals that nested $[*]$ modalities are not needed for *DEXPTIME*-hardness. But we can even do better, *i.e.*, restrict the language further.

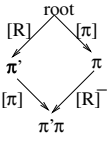
We define the fragment *flat core PDL$_n$* in Figure 2. A formula of flat core *PDL$_n$* is a conjunction of propositional variables and expressions of the form $[*]\,(p \leftrightarrow B)$. Note that $[*]$ modalities cannot be nested. Furthermore, all Boolean sub-formulas B are flat in that Boolean connectives only apply to variables.

Theorem 2. *Satisfiability of flat core PDL$_n$ formulas is DEXPTIME-complete.*

A proof is given in the full paper [21]. It is based on a new idea, by reduction to the emptiness of intersections of tree automata. This problem was shown *DEXPTIME*-hard by Seidl [29].

3.4 Inversion

We now consider a variant of *PDL$_n$* with inverted modalities $[R]^-$, which address all nodes $\pi'\pi$ reached by prefixing some $\pi' \in L(R)$ to the actual node π.



$$M, \pi \models [R^-]A \text{ if for all } \pi' \in L(R)\text{: } M, \pi'\pi \models A$$

Inverted flat core PDL$_n$ is defined in analogy to flat core *PDL$_n$* except that all modalities are inverted.

$$\begin{array}{ll} B ::= p_1 \wedge p_2 \mid \neg p \mid [i]^- p & \text{for } 1 \le i \le n \\ C ::= p \mid [*]\,(p \leftrightarrow B) \mid C_1 \wedge C_2 & \end{array}$$

We will freely omit inversion for $[*]$ operators, as these are never nested below modalities. We can translate flat core *PDL$_n$* formulas C into formulas C^- of the inverted flat core, and vice versa, by replacing the operators $[i]$ through $[i]^-$. Models can be inverted too: $M^-(p, \pi) = M(p, \pi^{-1})$ where π^{-1} is the inversion of π.

Lemma 1. $M \models C$ *iff* $M^- \models C^-$.

4 Uniform Subtype Satisfiability

We next investigate the complexity of uniform subtype satisfiability. We first show how to encode uniform subtype constraints into inverted *PDL$_n$*. We then give a translation from *inverted flat core PDL$_n$* back to uniform subtype satisfiability. Both translations are in polynomial time and preserve satisfiability (Propositions 2 and 3). The complexity of *PDL$_n$* (Theorem 2) thus carries over to uniform subtype satisfiability.

Theorem 3. *Uniform subtype satisfiability over possibly infinite trees is DEXPTIME-complete.*

4.1 Uniform Subtype Constraints into *PDL$_n$*

We encode uniform subtype constraints interpreted over infinite n-ary trees into inverted *PDL$_n$*. The translation relies on ideas of Tiuryn and Wand [33], but it is simpler with modal logics as the target language. We first present our translation for covariant uniform signatures and then sketch the contravariant case.

Table 3. Expressing uniform covariant subtype constraints in inverted PDL_n

$$
\begin{array}{rcl}
[\![x{=}f(x_1,\ldots,x_n)]\!] & =_{\mathrm{df}} & p_{x=f} \;\wedge\; \bigwedge_{g\in\Sigma}\bigwedge_{1\le i\le n}[*]\,(p_{x_i=g} \leftrightarrow [i]^- p_{x=g}) \\
[\![x{\le}y]\!] & =_{\mathrm{df}} & [*]\bigvee_{f\le_\Sigma g}(p_{x=f}\wedge p_{y=g}) \\
[\![\varphi_1\wedge\varphi_2]\!] & =_{\mathrm{df}} & [\![\varphi_1]\!]\wedge[\![\varphi_2]\!]
\end{array}
$$

Let Σ be a uniform covariant signature and $n > 1$ the arity of its function symbols. We fix a finite set of type variables V and consider subtype constraints φ over Σ with $V(\varphi) \subseteq V$. For all $x \in V$ and $f \in \Sigma$ we introduce propositional variables $p_{x=f}$ that are true at all nodes $\pi \in \{1,\ldots,n\}^*$ where the label of x is f.

The *well-formedness formula* wff_V states that all nodes of tree values of all $x \in V$ carry a unique label f: $\mathit{wff}_V =_{\mathrm{df}} \bigwedge_{x\in V}[*]\,(\dot{\bigvee}_{f\in\Sigma} p_{x=f})$. A polynomial time encoding of subtype constraints is presented in Table 3. Inverted modalities $[i]^-$ are needed to translate $x{=}f(x_1,\ldots,x_n)$ since $\alpha \models x{=}f(x_1,\ldots,x_n)$ if and only if $\alpha(x)(\varepsilon) = f$ and $\alpha(x)(i\pi) = \alpha(x_i)(\pi)$ for all words $i\pi \in \{1,\ldots,n\}^*$.

Proposition 1. *A uniform subtype constraint φ over a covariant signature Σ with $V(\varphi) \subseteq V$ is satisfiable if and only if $\mathit{wff}_V \wedge [\![\varphi]\!]$ is satisfiable.*

Proof. A solution of φ is a function $\alpha : V \to \mathit{tree}_\Sigma$. Let n be the arity of function symbols in Σ, so that all trees in tree_Σ are complete n-ary trees with nodes labeled in Σ, *i.e.*, total functions of type $\{1,\ldots,n\}^* \to \Sigma$. A variable assignment α thus defines a PDL_n model $M_\alpha : P \times \{1,\ldots,n\}^* \to \Sigma$ that satisfies for all $x \in V$ and $\pi \in \{1,\ldots,n\}^*$: $M_\alpha(p_{x=f},\pi) \leftrightarrow \alpha(x)(\pi) = f$. We can now show by induction on the structure of φ that $\alpha \models \varphi$ iff $M_\alpha, \varepsilon \models \mathit{wff}_V \wedge [\![\varphi]\!]$.

Proposition 2. *Uniform subtype satisfiability with covariant signatures over possibly infinite trees is in DEXPTIME.*

Proof. It remains to show that our reduction is in polynomial time. This might seem obvious, but it needs some care. Exclusive disjunctions of the form $p_1 \dot{\vee} \ldots \dot{\vee} p_n$ as used in the well-formedness formula can be encoded in quadratic time through $\bigvee_{i=1}^n (p_i \wedge \bigwedge_{1\le j\ne i\le n} \neg p_j)$. Equivalences $p \leftrightarrow \neg p'$ as used can be encoded in linear time by $(p \wedge \neg p') \vee (\neg p \wedge p')$.

Contravariance. Our approach smoothly extends to uniform subtyping with contravariant signatures. The key idea is that we can express polarities in *inverted* flat core PDL_n by using a new propositional variable p_{pol}. For example, consider the uniform signature $\Sigma = \{\to\}$, where $\to$ is the usual function type constructor. The variable p_{pol} is true in nodes with polarity 1 and false otherwise:

$$p_{pol} \wedge [*]\,(p_{pol} \leftrightarrow [1]^- \neg p_{pol}) \;\wedge\; [*]\,(p_{pol} \leftrightarrow [2]^- p_{pol}).$$

Limitation Due to Inversion. Inversion is crucial to our translation and has a number of consequences. Most importantly, we cannot express the formula $[*](p \to [+]p')$ in

Table 4. Boolean operations expressed by subtype constraints

$\mathit{all\text{-}c}(x)$	$=_{\mathrm{df}}$	$x{=}c(x,\ldots,x) \quad$ for some $c \in \Sigma(n)$
$\mathit{all\text{-}bool}(x)$	$=_{\mathrm{df}}$	$\exists y \exists z.\ \mathit{all}\text{-}0(x) \wedge x{\leq}y{\leq}z \wedge \mathit{all}\text{-}1(z)$
$\mathit{all\text{-}\overline{bool}}(x)$	$=_{\mathrm{df}}$	$\exists y \exists z.\ \mathit{all}\text{-}\bar{1}(x) \wedge x{\leq}y{\leq}z \wedge \mathit{all}\text{-}\bar{0}(z)$
$\mathit{upper}(x,y)$	$=_{\mathrm{df}}$	$\exists z.\ x{\leq}z \wedge y{\leq}z$
$\mathit{lower}(x,y)$	$=_{\mathrm{df}}$	$\exists z.\ z{\leq}x \wedge z{\leq}y$
$y{=}\overline{x}$	$=_{\mathrm{df}}$	$\mathit{all\text{-}bool}(x) \wedge \mathit{all\text{-}\overline{bool}}(y) \wedge \mathit{upper}(x,y) \wedge \mathit{lower}(x,y)$
$\mathit{all}(p_1 \vee p_2 \vee \neg p_3 \vee \neg p_4)$	$=_{\mathrm{df}}$	$\exists z.\ \bigwedge_{1\leq i\leq 4} \mathit{all\text{-}bool}(X_{p_i})$ $\wedge \mathit{lower}(z, X_{p_1}) \wedge \mathit{upper}(z, \overline{X_{p_2}})$ $\wedge \mathit{lower}(z, \overline{X_{p_3}}) \wedge \mathit{upper}(z, X_{p_4})$
$\mathit{all}(p_1 \vee p_2)$	$=_{\mathrm{df}}$	$\exists X_q.\ \mathit{all}(p_1 \vee p_2 \vee \neg q \vee \neg q) \wedge \mathit{all}\text{-}1(X_q)$

inverted *PDL*$_n$, which states that whenever p holds at some node then p' holds in all proper descendants.

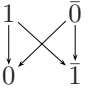
As a consequence, we cannot directly translate subtype constraints over standard signatures into *PDL*$_n$ (which we consider in Sections 5). The difficulty is to encode tree domains in the presence of leafs. Suppose we want to define that p holds for all nodes outside the tree domain. We could do so by imposing $[*](p_c \rightarrow [+]p)$ for all constants c, but this is impossible in inverted *PDL*$_n$.

This is not a problem for uniform signatures where every tree is completely n-ary, so that we do not need to express tree domains, as long as we are considering satisfiability. Unfortunately, however, the same technique does not extend to entailment and other fragments of first-order logic with negation.

4.2 Back Translation

To prove *DEXPTIME*-hardness of uniform subtype satisfiability, we show how to express inverted flat core *PDL*$_n$ by uniform subtype constraints, indeed only with covariant signatures. Our encoding of Boolean logic is inspired by Tiuryn [32], while the idea to lift this encoding to *PDL*$_n$ is new.

Let C be a formula of inverted flat core *PDL*$_n$. We aim to find a subtype constraints $[\![C]\!]^{-1}$ which preserves satisfiability. The critical point is how to translate *PDL*$_n$'s negation since it is absent in our target language of uniform subtype constraints. We work around by constructing a uniform subtype constraints with function symbols ordered in a crown: $\Sigma(n) = \{0, \bar{0}, 1, \bar{1}\}$.



All function symbols have arity n and satisfy $x \leq_{\Sigma(n)} y$ for all $x \in \{0, \bar{1}\}$, $y \in \{1, \bar{0}\}$. The symbols 0 and 1 model *PDL*$_n$'s underlying boolean lattice $\mathit{bool} = \{\mathit{0}, \mathit{1}\}$; the additional two symbols are introduced to define negation by $\mathit{neg}(c) = \overline{c}$ for $c \in \mathit{bool}$.

Next, Table 4 shows how to define *neg* by a subtype constraint. For every propositional variable p we introduce a new type variables X_p in the subtype constraint we are constructing to.

The subtype constraint *all-c*(x) holds for the unique trees that is completely labeled by some $c \in \Sigma(n)$. The subtype constraint *all-bool*(x) holds for trees that are labeled in $bool$. The constraints *lower*(x, y) and *upper*(x, y) require the existence of lower and upper bounds respectively for trees x and y. These bounds are used to define the diagonal pairs $y{=}\overline{x}$ in the crown.

Lemma 2. $y = \overline{x} \models \forall\pi.\ (x(\pi) = 0 \wedge y(\pi) = \bar{0}) \vee (x(\pi) = 1 \wedge y(\pi) = \bar{1})$.

Proof. Since x is a tree labeled in $bool$, all nodes π satisfy $\alpha(x)(\pi){=}0$ or $\alpha(x)(\pi){=}1$. In the first case (the second is analogous) the constraint *lower*(x, y) entails $\alpha(y)(\pi){\neq}\bar{1}$. Since y is a $\overline{bool}$ tree, $\alpha(y)(\pi){=}\bar{0}$.

Solutions of subtype constraints are variable assignments $\alpha : P \to \{1, \ldots, n\}^* \to \Sigma(n)$. For variable assignments α into trees over Booleans, we define corresponding *PDL*$_n$-models $M_\alpha : P \times \{1, \ldots, n\}^* \to bool$ by $M_\alpha(p, \pi) = \alpha(X_p)(\pi)$.

Lemma 3. *Let A be the Boolean formula $p_1 \vee p_2 \vee \neg p_3 \vee \neg p_4$. For all variable assignments α to trees over $\Sigma(n)$, $\alpha \models$ all(A) if and only if M_α is defined and $M_\alpha \models [*]A$.*

The lemma relies on a non-trivial property of the crown poset. For all $p_1, p_2, p_3, p_4 \in bool$:

$$p_1 \vee p_2 \vee \neg p_3 \vee \neg p_4 \models \exists z \in \{0, 1, \bar{0}, \bar{1}\}.\ \mathit{lower}(z, p_1) \wedge \mathit{upper}(z, \overline{p_2}) \wedge \mathit{lower}(z, \overline{p_3}) \wedge \mathit{upper}(z, p_4)$$

We illustrate the claim for $p_3 = p_4 = 1$ where the left hand side is equivalent to $p_1 \vee p_2$. The conjunction of the last two literals becomes $\mathit{lower}(z, \bar{1}) \wedge \mathit{upper}(z, 1)$ which is equivalent to $z \in \{1, \bar{1}\}$. The first two literals with $p_1 = p_2 = 0$ yields:

$$\mathit{lower}(z, 0) \Rightarrow z \neq \bar{1} \quad \text{and} \quad \mathit{upper}(z, \bar{0}) \Rightarrow z \neq 1$$

Thus, the complete conjunction is unsatisfiable with $p_1 = p_2 = 0$. Conversely, if $p_1 = 1$ then we can choose $z = \bar{1}$ since $\mathit{upper}(\bar{1}, \overline{p_2})$ holds for all $p_2 \in bool$. Similarly, if $p_2 = 1$ then we can choose $z = 1$ since $\mathit{lower}(1, p_1)$ for all $p_1 \in bool$.

The back translation $[\![C]\!]^{-1}$ of inverted flat core *PDL*$_n$ into subtype constraints is shown in Table 5. All Boolean formulas used there can be expressed by $p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4$ which we know how to encode.

Proposition 3. *Let C be a flat core inverted PDL$_n$ formula. For all variable assignments α to trees over $\Sigma(n)$, $\alpha \models [\![C]\!]^{-1}$ if and only if M_α is defined and $M_\alpha \models C$.*

For $n = 0$, subtype constraints become ordering constraints for a poset, while PDL_0 satisfiability becomes a Boolean satisfiability problem that is well-known to be NP-complete. We thus obtain a new NP-completeness proof for ordering constraints interpreted over posets [26].

5 Equivalence of Subtype Problems

We next show the equivalence of uniform subtype satisfiability with structural and non-structural subtype satisfiabilities over possibly infinite trees. Subtype satisfiability over finite trees will be treated in Section 6.

Table 5. Inverted core flat PDL_n in subtype constraints

$$
\begin{array}{rcl}
[\![p]\!]^{-1} & =_{\mathrm{df}} & \exists x_1 \ldots \exists x_m.\ \mathit{all\text{-}bool}(X_p) \wedge X_p{=}1(x_1, \ldots, x_m) \\
[\![[*]\,(p \leftrightarrow [i]^- q)]\!]^{-1} & =_{\mathrm{df}} & \mathit{all\text{-}bool}(X_p) \wedge \mathit{all\text{-}bool}(X_q) \\
 & & \wedge\, \exists x_1 \ldots \exists x_m.(0(x_1, ..., x_m){\leq}X_q{\leq}1(x_1, ..., x_m) \wedge X_p{=}x_i) \\
[\![[*]\,(p \leftrightarrow \neg q)]\!]^{-1} & =_{\mathrm{df}} & \mathit{all}(p \vee q) \wedge \mathit{all}(\neg p \vee \neg q) \\
[\![[*]\,(p \leftrightarrow (q_1 \wedge q_2))]\!]^{-1} & =_{\mathrm{df}} & \mathit{all}(\neg p \vee q_1) \wedge \mathit{all}(\neg p \vee q_2) \wedge \mathit{all}(p \vee \neg q_1 \vee \neg q_2) \\
[\![C_1 \wedge C_2]\!]^{-1} & =_{\mathrm{df}} & [\![C_1]\!]^{-1} \wedge [\![C_2]\!]^{-1}
\end{array}
$$

Theorem 4. *Structural, non-structural, and uniform subtype satisfiability over possibly infinite trees are equivalent and DEXPTIME-complete.*

The proof relies on constraints for subtype orders with a single nonconstants type constructor that we call 1-subtype orders.

1-subtype satisfiability is the satisfiability problem of subtype constraints over 1-subtype orders. This problem is parametric in the arities and polarities of the unique type constructor, the partial order on constants $(B, \leq_B)$, and whether or not $\{\bot, \top\}$ is included in the signature.

We present the proof in four steps. We first show how to reduce structural subtype satisfiability to 1-subtype satisfiability (Section 5.1) and then do the same for the non-structural case (Section 5.2). Next, we reduce 1-subtype satisfiability to uniform subtype satisfiability (Section 5.3). Finally, we translate uniform subtype satisfiability back to both structural and non-structural subtype satisfiability (Section 5.4).

5.1 Structural to 1-Subtype Satisfiability

In this part, we show how to reduce structural to 1-subtype satisfiability. We first use a standard technique to characterize the shapes of solutions to a structural subtype constraints. Given a constraint φ over Σ, we construct the *shape constraint* of φ, $sh(\varphi)$, by replacing each constant in φ with an arbitrary, fixed constant $\star \in \Sigma$, and each inequality with an equality:

$$
\begin{array}{rcl@{\qquad}rcl}
sh(x{=}f(x_1, x_2)) & =_{\mathrm{df}} & x{=}f(x_1, x_2), & sh(x{\leq}y) & =_{\mathrm{df}} & x{=}y, \\
sh(\varphi_1{\wedge}\varphi_2) & =_{\mathrm{df}} & sh(\varphi_1){\wedge}sh(\varphi_2), & sh(x{=}c) & =_{\mathrm{df}} & x{=}\star
\end{array}
$$

The constraint φ is called *weakly unifiable* iff $sh(\varphi)$ is unifiable.

Next, we handle contravariance. Consider a signature $\Sigma = B \cup \{\times, \rightarrow\}$. We construct a signature $s(\Sigma) =_{\mathrm{df}} B \cup \{f, c\}$, where f is function symbol of arity four and c is a fresh constant. Our approach is to use f to capture both $\times$ and $\rightarrow$, *i.e.*, all the non-constant function symbols in Σ. The first two arguments of f are used to model the two arguments of $\times$ and the next two to model the two arguments of $\rightarrow$. Thus, f is co-variant in all arguments except the third one.

Given a constraint φ over Σ, we construct $s(\varphi)$ over $s(\Sigma)$:

$$\begin{array}{rlrl} s(x{=}y{\times}z) &=_{\mathrm{df}} x{=}f(y,z,c,c), & s(x{=}y\to z) &=_{\mathrm{df}} x{=}f(c,c,y,z),\\ s(\varphi_1{\wedge}\varphi_2) &=_{\mathrm{df}} s(\varphi_1){\wedge}s(\varphi_2), & s(x{\leq}y) &=_{\mathrm{df}} x{\leq}y,\\ s(x{=}b) &=_{\mathrm{df}} x{=}b\ \forall b{\in}B \end{array}$$

Lemma 4. *If φ is weakly unifiable, then φ is satisfiable over Σ iff $s(\varphi)$ is satisfiable over $s(\Sigma)$.*

The proof of the above lemma requires the following result. Let φ be a constraint over a structural signature Σ. We have the following result due to Frey [8] that relates the shape of a solution of φ to that of a solution of $sh(\varphi)$.

Lemma 5 (Frey [8]). *If φ is satisfiable, let α be a solution of $sh(\varphi)$. Then φ has a solution β that is of the same shape as α,* i.e.*, for all $x \in V(\varphi) = V(sh(\varphi))$, $sh(\alpha(x) = \beta(x))$ is unifiable.*

5.2 Non-structural to 1-Subtype Satisfiability

We handle non-structural signatures $\Sigma = B \cup \{\bot, \top, \times, \to\}$, similarly. The new signature is defined in exactly the same way as for the structural case by $s(\Sigma) = B \cup \{\bot, \top, f, c\}$. Constraints are also transformed in the same way, except including two extra rules for $\bot$ and $\top$:

$$s(x{=}\bot) =_{\mathrm{df}} x{=}\bot, \quad s(x{=}\top) =_{\mathrm{df}} x{=}\top$$

However, weak unifiability is not sufficient for the initial satisfiability check. To see that, consider, for example, $x{\leq}y \times z \wedge x{\leq}u \to v$, which is satisfiable, but not weakly unifiable. To address this problem, we introduce a notion of *weak satisfiability*. It is similar to weak unfiability, except subtype ordering is also retained.

Definition 1. *Let φ be a constraint over Σ, and c be an arbitrary and fixed constant. Define the* weak satisfiability constraint $ws(\varphi)$ *as:*

$$\begin{array}{llll} ws(x{=}f(x_1,x_2)) =_{\mathrm{df}} x{=}f(x_1,x_2), & ws(x{\leq}y) =_{\mathrm{df}} x{\leq}y, & ws(x{=}\bot) =_{\mathrm{df}} x{=}\bot,\\ ws(\varphi_1{\wedge}\varphi_2) =_{\mathrm{df}} ws(\varphi_1){\wedge}ws(\varphi_2), & ws(x{=}b) =_{\mathrm{df}} x{=}c, & ws(x{=}\top) =_{\mathrm{df}} x{=}\top \end{array}$$

The constraint φ is called weakly satisfiable *iff $ws(\varphi)$ is satisfiable.*

Lemma 6. *If φ is weakly satisfiable, then φ is satisfiable over Σ iff $s(\varphi)$ is satisfiable over $s(\Sigma)$.*

The proof of this lemma requires the following result. Let φ be a constraint over a non-structural signature Σ. If $ws(\varphi)$ is satisfiable, then $ws(\varphi)$ has a minimum shape solution α by a simple extension of a theorem of Palsberg, Wand and OKeefe on non-structural subtype satisfiability over lattices [23]. We claim that if φ is satisfiable, then φ also has a minimum shape solution that is of the same shape as α.

Lemma 7. *If φ is satisfiable over Σ, let α be a minimum shape solution for $ws(\varphi)$, and in addition, α is such a solution with the least number of leaves assigned $\star$. Then φ has a solution β that is of the same shape as α,* i.e., *for all $x \in V(\varphi) = V(ws(\varphi))$, $sh(\alpha(x) = \beta(x))$ is unifiable. Furthermore, β is a minimum shape solution of φ.*

Lemma 5 and Lemma 7 together imply the following corollary, which is used next in Section 6 to treat subtype satisfiability interpreted over finite trees.

Corollary 1. *A subtype constraint φ is satisfiable over finite trees if and only if φ is satisfiable over finite trees of height bounded by $|\varphi|$. This holds for both structural and non-structural signatures.*

5.3 1-Subtype to Uniform Satisfiability

In this part, we give a reduction from 1-subtype to uniform subtype satisfiability. This reduction is uniform for subtyping with and without $\bot$ and $\top$.

Proposition 4. *Over possibly infinite trees, 1-subtype satisfiability is linear time reducible to uniform subtype satisfiability.*

Proof. Let Σ be a 1-subtype signature. We define a uniform signature $s(\Sigma)$ by extending the arities of all function symbols to the maximal arity of Σ (*i.e.*, the arity of the only non-trivial function symbol), such that: (1) $s(\Sigma) =_{\mathrm{df}} \Sigma$; (2) $\forall f \in s(\Sigma).arity_{s(\Sigma)}(f) =_{\mathrm{df}} \max$; and (3) $\leq_{s(\Sigma)} =_{\mathrm{df}} \leq_{\Sigma}$, where $\max$ is the maximal arity of Σ.

We next translate a subtype constraint φ over Σ to a constraint $s(\varphi)$ over $s(\Sigma)$:

$$
\begin{array}{ll}
s(x{=}f(x_1,\ldots,x_{\max})) =_{\mathrm{df}} x{=}f(x_1,\ldots,x_{\max}), & s(x{=}b) =_{\mathrm{df}} x{=}b(y_1,\ldots,y_{\max}),\\
s(\varphi_1 \wedge \varphi_2) =_{\mathrm{df}} s(\varphi_1) \wedge s(\varphi_2), & s(x_1 {\leq} x_2) =_{\mathrm{df}} x_1 {\leq} x_2,\\
s(x{=}\bot) =_{\mathrm{df}} x{=}\bot(u_1,\ldots,u_{\max}), & s(x{=}\top) =_{\mathrm{df}} x{=}\top(v_1,\ldots,v_{\max})
\end{array}
$$

where the y_i's, u_i's, and v_i's are fresh variables, and the last two rules are additional ones for a non-structural signature.

Lemma 8. *A subtype constraint φ over a standard signature Σ is satisfiable if and only if $s(\varphi)$ is satisfiable over the uniform signature $s(\Sigma)$.*

5.4 Uniform to (Non-)Structural Satisfiability

In this part, we prove the last step of the equivalence (Theorem 4), namely, how to reduce uniform satisfiability to structural and non-structural satisfiabilities.

Proposition 5. *Uniform subtype satisfiability is linear time reducible to structural and non-structural subtype satisfiability over possibly infinite trees.*

To simplify its proof we assume a uniform subtype problem where all function symbols have arity three with their first two arguments being contravariant and the last one covariant. This proof can be easily adapted to uniform signatures with other arities and polarities.

We construct a reverse translation $\bar{\bar{s}}$ of s (defined in Section 5.3) in two steps. Let Σ be a uniform signature with symbols of arity three. We first define a standard signature

$\bar{s}(\Sigma)$ by including symbols in Σ as constants and adding $\to$: (1) $\bar{s}(\Sigma) =_{\mathrm{df}} \Sigma \cup \{\to\}$; (2) $\forall g \in \Sigma.\mathit{arity}_{\bar{s}(\Sigma)}(g) =_{\mathrm{df}} 0$; (3) $\mathit{arity}_{\bar{s}(\Sigma)}(\to) =_{\mathrm{df}} 2$; and (4) $\leq_{\bar{s}(\Sigma)} =_{\mathrm{df}} \leq_{\Sigma}$. We now translate a subtype constraint φ over Σ to a constraint $\bar{s}(\varphi)$ over $\bar{s}(\Sigma)$:

$$\begin{aligned}\bar{s}(x{=}g(x_1,x_2,x_3)) &=_{\mathrm{df}} x{=}(x_3 \to x_2) \to (x_1 \to g)\\ \bar{s}(x_1 \leq x_2) &=_{\mathrm{df}} x_1 \leq x_2\\ \bar{s}(\varphi_1 \wedge \varphi_2) &=_{\mathrm{df}} \bar{s}(\varphi_1) \wedge \bar{s}(\varphi_2)\end{aligned}$$

where we use a non-flat constraint in the first line for a simpler presentation. The arguments x_1, x_2 are again contravariant and x_3 is covariant in the constraint $\bar{s}(x{=}g(x_1,x_2,x_3))$. Thus, $\bar{s}$ preserves all polarities.

In our second step, we force every variable to be mapped to a fixed, infinite shape. We extend $\bar{s}(\Sigma)$ to $\bar{\bar{s}}(\Sigma)$ with four new constants a_1, a_2, a_3, and a_4 with the following ordering: $a_1 \leq c \leq a_3 \ \wedge\ a_2 \leq c \leq a_4$, for all constants $c \in \bar{s}(\Sigma)$. We define $\bar{\bar{s}}(\varphi)$ as the conjunction of $\bar{s}(\Sigma)$ and the following constraints:

(1) $u_1 \leq x \wedge u_2 \leq x \wedge x \leq u_3 \wedge x \leq u_4$, for each variable $x \in V(\bar{s}(\Sigma))$;
(2) $\bigwedge_{i=1,2,3,4} u_i{=}(u_i \to u_i) \to (u_i \to a_i)$

The constraints (1) and (2) in $\bar{\bar{s}}(\varphi)$ determine the shape of any variable $x \in V(\bar{s}(\varphi))$. We claim, in the following lemma, that any solution to $\bar{\bar{s}}(\varphi)$ must be of a particular shape and must also map variables $x \in V(\bar{s}(\varphi))$ to trees over $\bar{s}(\Sigma)$.

Lemma 9. *If $\bar{\bar{s}}(\varphi)$ is interpreted over any (non-)structural signature $\bar{\bar{s}}(\Sigma)$ or $\bar{\bar{s}}(\Sigma) \cup \{\bot, \top\}$, any variable assignment $\alpha \models \bar{\bar{s}}(\varphi)$ satisfies for all paths $\pi \in (1(1\cup 2) \cup 21)^*$:*

$$\begin{aligned}\alpha(x)(\pi') &= \to && \text{if } \pi' \text{ is a prefix of } \pi\\ \alpha(x)(\pi 22) &= \begin{cases} a_i & \text{if } x = u_i\\ c \in \Sigma & \text{otherwise.}\end{cases}\end{aligned}$$

Lemma 10. *A subtype constraint φ over a uniform signature Σ is satisfiable if and only if the constraint $\bar{\bar{s}}(\varphi)$ over $\bar{\bar{s}}(\Sigma)$ is satisfiable. This statement also holds if we replace the structural signature $\bar{\bar{s}}(\Sigma)$ by the non-structural signature $\bar{\bar{s}}(\Sigma) \cup \{\bot, \top\}$.*

Proof. We define a transformation of $\mathit{map} : \mathit{tree}_{\Sigma} \to \mathit{tree}_{\bar{\bar{s}}(\Sigma)}$ on trees for all $g \in \Sigma$:

$$\begin{aligned}\mathit{map}(g(\tau_1,\tau_2,\tau_3)) =_{\mathrm{df}}\ & (\mathit{map}(\tau_3) \to \mathit{map}(\tau_2))\\ & \to (\mathit{map}(\tau_1) \to g)\end{aligned}$$

With that it can be easily verified that if there exists a solution $\alpha \models \varphi$ over an uniform signature Σ then $\mathit{map}(\alpha) \models \bar{\bar{s}}(\varphi)$ holds over $\bar{\bar{s}}(\Sigma)$. For the other direction we assume an assignment $\alpha \models \bar{\bar{s}}(\varphi)$. Then there also exists an assignment $\beta = \mathit{map}^{-1}(\alpha)$ according to the shape of any solution of $\bar{\bar{s}}(\varphi)$ stated in Lemma 9. Again, it can be easily verified that $\beta \models \Sigma$.

The proof also holds in the case where we add $\bot$ and $\top$ to $\bar{\bar{s}}(\Sigma)$ since both symbols cannot occur in any node of any solution of $\bar{\bar{s}}(\Sigma)$ (again Lemma 9).

6 Finite Subtype Satisfiability over Posets

Finite structural subtype satisfiability was shown *PSPACE*-complete by Tiuryn [32] and Frey [8]. Here, we establish the same complexity for the non-structural case.

Proposition 6. *Non-structural subtype satisfiability over finite trees is PSPACE-hard.*

The analogous result for the structural case was shown by Tiuryn [32]). To lift this result, we show how to reduce non-structural to structural subtype satisfiability.

Lemma 11. *Structural subtype satisfiability is polynomial time reducible to non-structural subtype satisfiability (both for finite and infinite trees).*

Proof. Let Σ be a structural signature. We construct a non-structural signature: $s(\Sigma) =_{\mathrm{df}} \Sigma \cup \{\bot, \top, a_1, a_2, a_3, a_4\}$ with the a_i's being four new constants. In addition, $\leq_{s(\Sigma)} =_{\mathrm{df}} \leq_\Sigma \cup \{(a_1, c), (a_2, c), (c, a_3), (c, a_4) \mid c \in \Sigma_0\}$.

Let φ be a constraint over Σ. We construct $s(\varphi)$ over $s(\Sigma)$. Consider φ's shape constraint $sh(\varphi)$ (see Section 5.1). If $sh(\varphi)$ is not unifiable, we simply let $s(\varphi) =_{\mathrm{df}} \top \leq \bot$. Otherwise, consider the most general unifier (m.g.u.) γ of $sh(\varphi)$. We let $sh(\varphi)'$ be the same as $sh(\varphi)$ except each occurrence of $\star$ is replaced with a fresh variable. We make two copies of $sh(\varphi)'$, $sh(\varphi)'_L$ and $sh(\varphi)'_R$ (for left and right), where each variable x is distinguished as x_L and x_R respectively. For each variable $x \in V(\varphi)$, if $\gamma(x)$ is either $\star$ or belongs to $V(\varphi)$, we say x is atomic. For a variable x, let $force(x)$ denote the constraint: $a_1 \leq x \wedge a_2 \leq x \wedge x \leq a_3 \wedge x \leq a_4$. Notice that Lemma 11 holds both for finite and infinite trees.

We can now construct $s(\varphi)$, which is the conjunction of the following components: (1) φ itself; (2) $sh(\varphi)'_L$; (3) $sh(\varphi)'_R$; (4) For each atomic $x \in V(\varphi)$, $force(x_L)$ and $force(x_R)$; (5) For each fresh variable x in $sh(\varphi)'_L$ and $sh(\varphi)'_R$, $force(x)$; and (6) For each variable $x \in V(\varphi)$, $x_L \leq x \leq x_R$. One can show that φ is satisfiable over Σ iff $s(\varphi)$ is satisfiable over $s(\Sigma)$.

By adapting the proof of Frey [8], we can show membership in *PSPACE*, and thus we have the following theorem. For an alternative proof of using *K-normal modal logic*, please refer to the full paper [21].

Theorem 5. *Finite non-structural subtype satisfiability is PSPACE-complete.*

7 Conclusions

We have given a complete characterization of the complexity of subtype satisfiability over posets through a new connection of subtype satisfiability with modal logics, which have well understood satisfiability problems. Our technique yields a uniform and systematic treatment of different choices of subtype orderings: finite versus recursive types, structural versus non-structural subtyping, and considerations of symbols with co- and contra-variant arguments.

Our technique, however, does not extend beyond satisfiability to other first-order fragments that require negations, such as subtype entailment, whose decidability is a

longstanding open problem over non-structural signatures. Negations can certainly be modeled by our modal logic, but only over uniform signatures. In fact, there must not exist reductions from standard signatures to uniform ones that preserve subtype entailment, for example. Otherwise, such a reduction would have implied that the first-order theory of non-structural subtyping, which is undecidable [31], were a fragment of S2S, which is decidable [27].

References

1. R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.
2. M. Ben-Ari, J. Y. Halpern, and A. Pnueli. Deterministic propositional dynamic logic: Finite models, complexity, and completeness. *Journal of Computer and System Sciences*, 25(3):402–417, 1982.
3. P. Blackburn, B. Gaiffe, and M. Marx. Variable free reasoning on finite trees. In *Proceedings of Mathematics of Language*, pages 17–30, 2003.
4. P. Blackburn and W. Meyer-Viol. Linguistics, logic, and finite trees. *Logic Journal of the IGPL*, 2:3–29, 1994.
5. J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA*, pages 169–184, 1995.
6. M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18:194–211, 1979.
7. C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, 1999.
8. A. Frey. Satisfying subtype inequalities in polynomial space. *Theoretical Computer Science*, 277:105–117, 2002.
9. Y. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, 1990.
10. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, 2000.
11. F. Henglein and J. Rehof. The complexity of subtype entailment for simple types. In *LICS*, pages 362–372, 1997.
12. F. Henglein and J. Rehof. Constraint automata and the complexity of recursive subtype entailment. In *ICALP*, pages 616–627, 1998.
13. M. Hoang and J. Mitchell. Lower bounds on type inference with subtypes. In *POPL*, 176–185, 1995.
14. D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994.
15. M. Kracht. Inessential features. In *Logical Aspects of Computational Linguistics*, volume 1328, pages 43–62, 1997.
16. V. Kuncak and M. Rinard. Structural subtyping of non-recursive types is decidable. In *LICS*, pages 96–107, 2003.
17. J. Mitchell. Coercion and type inference. In *POPL*, pages 175–185, 1984.
18. J. C. Mitchell. Type inference with simple subtypes. *The Journal of Functional Programming*, 1(3):245–285, 1991.
19. J. Niehren and T. Priesnitz. Entailment of non-structural subtype constraints. In *Asian Computing Science Conference*, pages 251–265, 1999.
20. J. Niehren and T. Priesnitz. Non-structural subtype entailment in automata theory. *Information and Computation*, 186(2):319–354, 2003.

21. J. Niehren, T. Priesnitz, and Z. Su. Complexity of subtype satisfiability over posets. Available at `www.ps.uni-sb.de/papers`, 2005.
22. A. Palm. Propositional tense logic for trees. In *Proceedings of the Sixth Meeting on Mathematics of Language*, pages 74–87, 1999.
23. J. Palsberg, M. Wand, and P. O'Keefe. Type Inference with Non-structural Subtyping. *Formal Aspects of Computing*, 9:49–67, 1997.
24. F. Pottier. Simplifying subtyping constraints. In *ICFP*, pages 122–133, 1996.
25. F. Pottier. *Type inference in the presence of subtyping: from theory to practice*. PhD thesis, INRIA, 1998.
26. V. Pratt and J. Tiuryn. Satisfiability of inequalities in a poset. *Fundamenta Informaticae*, 28:165–182, 1996.
27. M. O. Rabin. Decidability of second-order theories and automata on infinte trees. *Trans. Amer. Math. Soc.* , 141:1–35, 1969.
28. J. Rehof. *The Complexity of Simple Subtyping Systems*. PhD thesis, DIKU, 1998.
29. H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60, 1994.
30. E. Spaan. *Complexity of Modal Logics*. PhD thesis, University of Amsterdam, 1993.
31. Z. Su, A. Aiken, J. Niehren, T. Priesnitz, and R. Treinen. First-order theory of subtyping constraints. In *POPL*, pages 203–216, 2002.
32. J. Tiuryn. Subtype inequalities. In *LICS*, pages 308–315, 1992.
33. J. Tiuryn and M. Wand. Type reconstruction with recursive types and atomic subtyping. In *Theory and Practice of Software Development*, 686-701,'93.
34. V. Trifonov and S. Smith. Subtyping constrained types. In *SAS*, pages 349–365, 1996.
35. M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal Computer & System Science*, 32(2):183–221, 1986.

A Type System Equivalent to a Model Checker

Mayur Naik[1] and Jens Palsberg[2]

[1] Stanford University
mhn@cs.stanford.edu
[2] UCLA
palsberg@ucla.edu

Abstract. Type systems and model checking are two prevalent approaches to program verification. A prominent difference between them is that type systems are typically defined in a syntactic and modular style whereas model checking is usually performed in a semantic and whole-program style. This difference between the two approaches lends them complementary to each other: type systems are good at explaining why a program was accepted while model checkers are good at explaining why a program was rejected.

We present a type system that is equivalent to a model checker for verifying temporal safety properties of imperative programs. The model checker is natural and may be instantiated with any finite-state abstraction scheme such as predicate abstraction. The type system which is also parametric type checks exactly those programs that are accepted by the model checker. It uses function types to capture flow sensitivity and intersection and union types to capture context sensitivity. Our result sheds light on the relationship between the two approaches, provides a methodology for studying their relative expressiveness, is a step towards sharing results between them, and motivates synergistic program analyses involving interplay between them.

1 Introduction

1.1 Background

Type systems and model checking are two prevalent approaches to program verification. It is well known that both approaches are essentially abstract interpretations and are therefore closely related [10, 11]. Despite deep connections, however, a prominent difference between them is that type systems are typically defined in a syntactic and modular style, using one type rule per syntactic construct, whereas model checking is usually performed in a semantic and whole-program style, by exploring the reachable state-space of a model of the program. This difference between type systems and model checking has a significant consequence: it lends the approaches complementary to each other, namely, type systems are better at explaining why a program was accepted whereas model checkers are better at explaining why a program was rejected.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 374–388, 2005.

A type inference algorithm that accepts a program annotates it with *types* (keywords: syntactic, modular) explaining why it was accepted. The benefits of type annotations are well known: they aid in understanding, modifying, reusing and certifying the program. However, it is often unnatural to explain why a program was rejected by a type inference algorithm, and there is a large body of work on explaining the source of type errors especially in the context of type inference algorithms for languages with higher-order functions like Haskell, Miranda, and ML [46, 24, 6, 14, 44, 9, 19] and, more recently, for languages with concurrency like Java [15, 16].

On the other hand, a model checker that rejects a program provides a *counterexample* which is a program trace (keywords: semantic, whole-program) that explains why the program was rejected. The benefits of counterexamples are well known: they aid in debugging the program. However, it is often unnatural to explain why a program was accepted by a model checker, and several proof systems for model checkers have been devised [39, 31, 38, 21, 43, 32].

This complementary nature of type systems and model checking motivates investigating the relationship between the two approaches, devising a methodology for studying their relative expressiveness, sharing results between them, and designing synergistic program analyses involving an interplay between a type system and a model checker.

1.2 Our Result

In this paper, we present a type system that is equivalent to a model checker for verifying temporal safety properties of imperative programs. In model checking terminology, a safety property is a temporal property whose violation can be witnessed by a finite program trace or, equivalently, by the failure of an assertion at a program point. Our model checker is conventional and may be instantiated with any finite-state abstraction scheme such as predicate abstraction [18]. The type system which is also parametric type checks exactly those programs that are accepted by the model checker. It uses function types to capture flow sensitivity and intersection and union types to capture context sensitivity.

The implications of our result may be summarized as follows:

1. Our work sheds light on the relationship between type systems and model checking. In particular, it shows that the most straightforward form of model checking corresponds to the most complex form of typing.
   Finite-state model checkers routinely associate with each statement s of the program a set of the form:

   $$\{\ \langle \omega_i, \omega_j \rangle \mid \omega_j \in \delta_s(\omega_i)\ \}$$

   where ω ranges over a finite set of abstract contexts Ω and $\delta_s : \Omega \rightharpoonup 2^{\Omega}$ is a partial function called the abstract transfer function associated with s. Intuitively, the above set says that if s begins executing in abstract context ω_i then it will finish executing in an abstract context $\omega_j \in \delta_s(\omega_i)$. For example, in model checkers such as SLAM [4], BLAST [22], and MAGIC [7], Ω

represents the set of all valuations to the finite set of predicates with respect to which the predicate abstraction (model) of the program is constructed.

Likewise, our type system assigns to each statement in the program, a finitary polymorphic type of the form:

$$\bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j)$$

where A and $\forall i \in A : B_i$ are finite. This is the most complex form of typing. Conventional type systems employ restricted cases of this form of typing such as ones requiring $|A| = 1$ (no intersection types) or $\forall i \in A : |B_i| = 1$ (no union types).

2. Our work provides a methodology for studying the relative expressiveness of a type system and a model checker. Our technique for proving the equivalence is novel and general: we have successfully applied it in two additional settings, namely, stack-size analysis [25] and deadline analysis [29] for a class of real-time programs called interrupt-driven programs [35].
3. Our work is a step towards sharing of results between the type systems and model checking communities. The backward direction of our equivalence theorem states that if the model checker accepts a program, then the program is well-typed. We prove this by building a type derivation from the model constructed by the model checker. We thereby obtain a model-checking-based type inference algorithm for our type system.
4. Our work motivates synergistic program analyses involving interplay between a type system and a model checker. The analyses can use types to document correct programs and counterexamples to explain erroneous programs. Moreover, they can be implemented efficiently due to the correspondence between types and models: types already existing in the program or inferred by a type inference algorithm can be used to construct a model for performing model checking, as illustrated in [12, 8], and conversely, a model constructed by a model checker can be used to infer types, as shown in this paper.

1.3 Proof Architecture

We present an overview of our technique for proving the equivalence. A typical type soundness theorem states that *well-typed programs do not go wrong* [27]. Usually, *going wrong* is formalized as *getting stuck* in the operational semantics. More formally, for a program s, an initial concrete environment σ, and an initial abstract environment ω, type soundness states that:

If $\langle s, \omega \rangle$ is well-typed then $\langle s, \sigma \rangle$ does not go wrong (in the concrete semantics).

Type checking requires a predefined set of abstractions, namely, the types. Then, the existence of a derivable type judgment implies that the program has the desired property. Model checking, on the other hand, is not concerned with types. It works with a model, that is, an abstract semantics, and can answer questions such as:

$\langle s, \omega \rangle$ does not go wrong (in the abstract semantics).

Model-checking soundness then states that:

> If $\langle s, \omega \rangle$ does not go wrong (in the abstract semantics) then
> $\langle s, \sigma \rangle$ does not go wrong (in the concrete semantics).

Our equivalence result states that:

> $\langle s, \omega \rangle$ is well-typed iff $\langle s, \omega \rangle$ does not go wrong (in the abstract semantics).

We prove the forward direction using a variant of type soundness in which the step relation is the abstract semantics instead of the concrete semantics and we prove the backward direction constructively by building a type derivation from the model constructed by the model checker.

It is important to note that we do not prove the soundness of either the type system or the model checker. Our equivalence result guarantees that the type system is sound iff the model checker is sound but it does not prevent both from being unsound. Proving soundness would require us to define a concrete semantics and to instantiate the type system and the model checker (recall that both are parametric). This in turn would detract from the generality of our equivalence result.

1.4 Rest of the Paper

In Section 2, we present an imperative WHILE language and a model checker for verifying temporal safety properties expressed as assertions in that language. In Section 3, we present a type system that is equivalent to the model checker. In Section 4, we prove the equivalence result. In Section 5, we illustrate the equivalence by means of examples. In Section 6, we discuss related work. Finally, in Section 7, we conclude with a note on future work.

2 Model Checker

The abstract syntax of our imperative WHILE language is as follows:

$$\text{(stmt)}\quad s ::= p \mid \texttt{assume}(e) \mid \texttt{assert}(e) \mid s_1;\ s_2 \mid \texttt{if}\ (*)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ (*)\ \texttt{do}\ s'$$

A statement s is either a primitive statement p (for instance, an assignment statement or a skip statement), an assume statement, an assert statement, a sequential composition of statements, a branching statement, or a looping statement. For the sake of generality, we leave primitive statements p and boolean expressions e uninterpreted. Our abstract syntax for branching and looping statements is standard in the literature on model checking. It is related to the more familiar syntax for these statements as follows:

$$\texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \equiv \texttt{if}\ (*)\ \texttt{then}\ \{\ \texttt{assume}(e);\ s_1\ \}\ \texttt{else}\ \{\ \texttt{assume}(\bar{e});\ s_2\ \}$$
$$\texttt{while}\ (e)\ \texttt{do}\ s' \equiv \{\ \texttt{while}\ (*)\ \texttt{do}\ \{\ \texttt{assume}(e);\ s'\ \}\ \};\ \texttt{assume}(\bar{e})$$

$$\text{(state)}\ \ a \ ::=\ \omega \ \mid\ \mathsf{error} \ \mid\ \langle s, \omega \rangle$$

$$\langle p, \omega_k \rangle \hookrightarrow \omega_l \qquad \text{if } l \in \delta_p(k) \tag{1}$$

$$\langle \mathtt{assume}(e), \omega_k \rangle \hookrightarrow \omega_k \qquad \text{if } k \in \delta_e \tag{2}$$

$$\langle \mathtt{assume}(e), \omega_k \rangle \hookrightarrow \mathsf{error} \qquad \text{if } k \notin \delta_e \tag{3}$$

$$\langle \mathtt{assert}(e), \omega_k \rangle \hookrightarrow \omega_k \qquad \text{if } k \in \delta_e \tag{4}$$

$$\frac{\langle s_1, \omega \rangle \hookrightarrow \omega'}{\langle s_1; s_2, \omega \rangle \hookrightarrow \langle s_2, \omega' \rangle} \tag{5} \qquad \frac{\langle s_1, \omega \rangle \hookrightarrow \mathsf{error}}{\langle s_1; s_2, \omega \rangle \hookrightarrow \mathsf{error}} \tag{6} \qquad \frac{\langle s_1, \omega \rangle \hookrightarrow \langle s_1', \omega' \rangle}{\langle s_1; s_2, \omega \rangle \hookrightarrow \langle s_1'; s_2, \omega' \rangle} \tag{7}$$

$$\langle \mathtt{if}\ (*)\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2, \omega \rangle \hookrightarrow \langle s_1, \omega \rangle \tag{8}$$

$$\langle \mathtt{if}\ (*)\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2, \omega \rangle \hookrightarrow \langle s_2, \omega \rangle \tag{9}$$

$$\langle \mathtt{while}\ (*)\ \mathtt{do}\ s', \omega \rangle \hookrightarrow \langle s';\ \mathtt{while}\ (*)\ \mathtt{do}\ s', \omega \rangle \tag{10}$$

$$\langle \mathtt{while}\ (*)\ \mathtt{do}\ s', \omega \rangle \hookrightarrow \omega \tag{11}$$

Fig. 1. Abstract Semantics

where $(*)$ denotes non-deterministic choice and $\bar{e}$ denotes the negation of e.

We next present a model checker for verifying temporal safety properties of programs expressed in our language. The class of temporal safety properties is precisely the class of properties whose violation can be witnessed by a finite program trace or, equivalently, by the failure of an assertion at a program point. Our model checker is conventional and is parameterized by the following components:

- A finite set of abstract contexts Ω.
- An abstract transfer function $\delta_p \in \Omega \to 2^\Omega$ per primitive statement p describing the effect of p on abstract contexts. We assume that δ_p is total and $\forall i \in \Omega : \delta_p(i) \neq \emptyset$.
- A predicate $\delta_e \subseteq \Omega$ per boolean expression e denoting the set of abstract contexts in which e is true.

These components may be instantiated by any finite-state abstraction scheme. For instance, if the scheme is predicate abstraction, then Ω is the set of all valuations to the finite set of predicates with respect to which the predicate abstraction of the program is constructed. For convenience, we treat Ω as a set of indices instead of abstract contexts. We use $i, j, ...$ to range over Ω and $\omega_i, \omega_j, ...$ to denote the corresponding abstract contexts indexed by them.

The abstract semantics of the model checker is presented in Figure 1. State $\langle s, \omega \rangle$ is *stuck* if $\nexists a : \langle s, \omega \rangle \hookrightarrow a$. The only kind of state that can get stuck is of the form $\langle \mathtt{assert}(e), \omega \rangle$ such that $\omega \notin \delta_e$. State $\langle s, \omega \rangle$ *goes wrong* if $\exists \langle s', \omega' \rangle : (\langle s, \omega \rangle \hookrightarrow^* \langle s', \omega' \rangle$ and $\langle s', \omega' \rangle$ is stuck). Given a program s and an abstract context ω, the model checker determines whether $\langle s, \omega \rangle$ goes wrong. If $\langle s, \omega \rangle$ goes wrong, it reports a counterexample which is a finite trace $\langle s, \omega \rangle \hookrightarrow^*$

$\langle \texttt{assert}(e), \omega' \rangle$ where $\omega' \notin \delta_e$. Otherwise, it returns the finite set of reachable abstract states $\{ a \mid \langle s, \omega \rangle \hookrightarrow^* a \}$ which serves as a proof that the concrete program does not go wrong, provided the model checker is sound. Model checking soundness is typically proved by showing that the abstract semantics simulates the concrete semantics (see for example [29, 25]).

3 Type System

Our type system assigns a type of the form $\bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j)$ to each statement in the program, where A and $\forall i \in A : B_i$ are subsets of Ω. Recall that Ω is finite whence the type is finitary. Intuitively, the type states that it is safe to begin executing the statement in one of contexts $\{ \omega_i \mid i \in A \}$ and, furthermore, if it begins executing in context ω_i $(i \in A)$ then it will finish executing in one of contexts $\{ \omega_j \mid j \in B_i \}$. Our type system includes the type $\top \triangleq \bigwedge \emptyset$ to handle the case in which A is empty, and the type $\bot \triangleq \bigvee \emptyset$ to handle the case in which any B_i $(i \in A)$ is empty.

The type rules are shown in Figure 2. We say that an abstract state $\langle s, \omega_k \rangle$ is *well-typed* if statement s can be assigned a type that states that it is safe to begin executing s in abstract context ω_k (see rule (12)).

Rule (13) type checks primitive statement p. The type of p captures the effect of the abstract transfer function δ_p associated with p. The side-condition of the rule states that it is safe to begin executing p in any context in Ω because we have assumed that δ_p is a total function.

Rule (14) type checks statement $\texttt{assume}(e)$. The side-condition of the rule says that it is safe to begin executing $\texttt{assume}(e)$ in any context in Ω and, moreover, the first conjunct in its type states that it has the effect of a skip statement if it begins executing in a context in which e is true while the second conjunct in its type states that there does not exist any context in which it finishes executing if it begins executing in a context in which e is false.

Rule (15) type checks statement $\texttt{assert}(e)$. The side-condition of the rule says that it is safe to begin executing $\texttt{assert}(e)$ only in a context in which e is true, and its type states that it has the effect of a skip statement if it begins executing in such a context.

Rule (16) type checks sequentially composed statements. The side-condition says that it is safe to begin executing s_1; s_2 only in contexts in which it is safe to begin executing s_1 and, moreover, if s_1 begins executing in such a context, then it must be safe to begin executing s_2 in each context in which s_1 might finish executing.

Rule (17) type checks branching statements. The side-condition says that it is safe to begin executing $\texttt{if}\ (*)\ \texttt{then}\ s_1\ \texttt{else}\ s_2$ only in contexts in which it is safe to begin executing both s_1 and s_2.

Rule (18) type checks looping statements. The side-condition says that it is safe to begin executing $\texttt{while}\ (*)\ \texttt{do}\ s'$ only in contexts in which it is safe to begin executing s' and, moreover, if s' begins executing in such a context, then it must be safe to begin executing $\texttt{while}\ (*)\ \texttt{do}\ s'$ in each context in which

$$\frac{s \;:\; \bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j)}{\langle s, \omega_k \rangle \text{ is well-typed}} \quad [k \in A] \tag{12}$$

$$p \;:\; \bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{j \in \delta_p(i)} \omega_j) \quad [A \subseteq \Omega] \tag{13}$$

$$\texttt{assume}(e) \;:\; \bigwedge_{i \in A}(\omega_i \rightarrow \omega_i) \;\wedge\; \bigwedge_{i \in B}(\omega_i \rightarrow \bot) \quad [A \subseteq \delta_e \wedge B \subseteq \Omega \setminus \delta_e] \tag{14}$$

$$\texttt{assert}(e) \;:\; \bigwedge_{i \in A}(\omega_i \rightarrow \omega_i) \quad [A \subseteq \delta_e] \tag{15}$$

$$\frac{\begin{array}{c} s_1 \;:\; \bigwedge_{i \in A_1}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j) \\ s_2 \;:\; \bigwedge_{i \in A_2}(\omega_i \rightarrow \bigvee_{j \in B'_i} \omega_j) \end{array}}{s_1; s_2 \;:\; \bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{k \in \bigcup\{\, B'_j \,\mid\, j \in B_i \,\}} \omega_k)} \quad \left[A \subseteq A_1 \wedge \bigcup_{i \in A} B_i \subseteq A_2\right] \tag{16}$$

$$\frac{\begin{array}{c} s_1 \;:\; \bigwedge_{i \in A_1}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j) \\ s_2 \;:\; \bigwedge_{i \in A_2}(\omega_i \rightarrow \bigvee_{j \in B'_i} \omega_j) \end{array}}{\texttt{if}\ (*)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \;:\; \bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{j \in B_i \cup B'_i} \omega_j)} \quad [A \subseteq A_1 \cap A_2] \tag{17}$$

$$\frac{s' \;:\; \bigwedge_{i \in A'}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j)}{\texttt{while}\ (*)\ \texttt{do}\ s' \;:\; \bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{k \in \mu X.(\{i\} \cup \{B_j \mid j \in X\})} \omega_k)} \quad \left[A \subseteq A' \wedge \bigcup_{i \in A} B_i \subseteq A\right] \tag{18}$$

$\mu X.E$ denotes the least fixed point of function $\lambda X.E : 2^{\Omega} \rightarrow 2^{\Omega}$

Fig. 2. Type Rules

s' might finish executing. Let $\mu X.E$ denote the least fixed point of the function $\lambda X.E : 2^{\Omega} \rightarrow 2^{\Omega}$. Then, the type of `while` $(*)$ `do` s' states that if the loop begins executing in context ω_i $(i \in A)$, then it will finish executing in one of contexts $\{\, \omega_k \mid k \in \mu X\,.\,(\{i\} \cup \{B_j \mid j \in X\})\,\}$, that is: (i) in the base case (0 iterations) the loop will finish executing in the context ω_i in which it began executing, and (ii) in the inductive case ($n+1$ iterations where $n \geq 0$) the loop will finish executing in one of contexts $\{\, \omega_k \mid k \in B_j \,\}$ where ω_j is a context in which the loop might finish executing in n iterations, in which case in the $n+1^{th}$ iteration, s' will begin executing in context ω_j and finish executing in one of contexts $\{\, \omega_k \mid k \in B_j \,\}$.

4 Equivalence

In this section, we prove that a program type checks if and only if the model checker accepts it.

The proof from type checking to model checking is similar to that of type soundness, consisting of Progress (Lemma 1) and Type Preservation (Lemma 2), the key difference being that the step relation is the abstract semantics of the model checker instead of the concrete semantics of the language.

Lemma 1. (Progress)
If $\langle s, \omega_m \rangle$ *is well-typed then* $\langle s, \omega_m \rangle$ *is not stuck.*

Proof. See Appendix.

Lemma 2. (Type Preservation)
If $\langle s, \omega_m \rangle$ *is well-typed and* $\langle s, \omega_m \rangle \hookrightarrow^t \langle s', \omega_n \rangle$ *then* $\langle s', \omega_n \rangle$ *is well-typed.*

Proof. See Appendix.

It is then straightforward to prove that if a program type checks then the model checker accepts it.

Lemma 3. (From Type Checking to Model Checking)
If $\langle s, \omega_m \rangle$ *is well-typed then* $\langle s, \omega_m \rangle$ *does not go wrong.*

Proof. Suppose $\langle s, \omega_m \rangle$ is well-typed. We need to prove that $\langle s, \omega_m \rangle \hookrightarrow^t \langle s', \omega_n \rangle$ implies $\langle s', \omega_n \rangle$ is not stuck. Suppose $\langle s, \omega_m \rangle \hookrightarrow^t \langle s', \omega_n \rangle$. From $\langle s, \omega_m \rangle$ is well-typed and $\langle s, \omega_m \rangle \hookrightarrow^t \langle s', \omega_n \rangle$ and lemma (2), we have $\langle s', \omega_n \rangle$ is well-typed. From $\langle s', \omega_n \rangle$ is well-typed and lemma (1), we have $\langle s', \omega_n \rangle$ is not stuck.

The proof from model checking to type checking is constructive and involves building a type derivation from the model constructed by the model checker. The following definitions show how to construct types from the model.

Definition 1. $\mathbb{A}^s = \{\, i \in \Omega \mid \langle s, \omega_i \rangle \textit{ does not go wrong} \,\}$

Definition 2. *Given statement* s *and* $i \in \Omega$*, define* $\mathbb{B}^{s,i} \subseteq \Omega$ *as follows:*

$$
\begin{array}{ll}
\mathbb{B}^{s,i} = \delta_p(i) & \textit{if } s = p \\
\mathbb{B}^{s,i} = \{i\} & \textit{if } s = \texttt{assume}(e) \textit{ or } \texttt{assert}(e) \textit{ and } i \in \delta_e \\
\mathbb{B}^{s,i} = \emptyset & \textit{if } s = \texttt{assume}(e) \textit{ or } \texttt{assert}(e) \textit{ and } i \notin \delta_e \\
\mathbb{B}^{s,i} = \bigcup \{\, \mathbb{B}^{s_2,j} \mid j \in \mathbb{B}^{s_1,i} \,\} & \textit{if } s = s_1;\ s_2 \\
\mathbb{B}^{s,i} = \mathbb{B}^{s_1,i} \cup \mathbb{B}^{s_2,i} & \textit{if } s = \texttt{if } (*) \texttt{ then } s_1 \texttt{ else } s_2 \\
\mathbb{B}^{s,i} = \mu X \,.\, (\{i\} \cup \{\, \mathbb{B}^{s',j} \mid j \in X \,\}) & \textit{if } s = \texttt{while } (*) \texttt{ do } s'
\end{array}
$$

The key lemma involves showing that the constructed types yield a valid type derivation. It is proved by induction on the structure of the program.

Lemma 4. (Typability) $s : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.

Proof. See Appendix.

It is then straightforward to prove that if a program is accepted by the model checker then it type checks.

Lemma 5. (From Model Checking to Type Checking)
If $\langle s, \omega_m \rangle$ *does not go wrong then* $\langle s, \omega_m \rangle$ *is well-typed.*

Proof. From lemma (4), we have $s : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$. From $\langle s, \omega_m \rangle$ does not go wrong and defn. (1), we have $m \in \mathbb{A}^s$. From $s : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$ and $m \in \mathbb{A}^s$ and rule (12), we have $\langle s, \omega_m \rangle$ is well-typed.

Finally, we present our main result which states that a program type checks if and only if the model checker accepts it.

Theorem 1. (Equivalence)
$\langle s, \omega \rangle$ is well-typed if and only if $\langle s, \omega \rangle$ does not go wrong.

Proof. Combine lemma (3) and lemma (5).

5 Examples

In this section, we illustrate our equivalence result by means of three examples.
Example 1. Consider the following program:

$$s_1 \triangleq \text{lock}_1(); \text{lock}_2() \quad \text{where} \quad \text{lock}() \triangleq \texttt{assert}(s = \texttt{U}); s := \texttt{L}$$

where U and L denote the unlocked and locked states, respectively. Suppose the model checker is instantiated with predicate abstraction in which case Ω is a set of program predicates, say $\{\texttt{s=U}, \texttt{s=L}\}$. It is easy to see that state $\langle s_1, \texttt{s=U} \rangle$ goes wrong in the abstract semantics of Figure 1 and is not well-typed in the type system of Figure 2. As a result, both the model checker and the type system reject it.

Notice that although not every state $\langle s, \omega \rangle$ is well-typed in our type system, every statement s is typable (see lemma (4)). For instance, although $\langle s_1, \texttt{s=U} \rangle$ is not well-typed, s_1 has the type $\top$. The following example motivates the need for making every statement typable, namely, the need for the type $\top$.
Example 2. Consider the following program:

$$s_2 \triangleq \text{lock}_1(); \texttt{assume}(\mathit{false}); \text{lock}_2()$$

Assuming the same predicate abstraction as in the previous example, it is easy to see that state $\langle s_2, \texttt{s=U} \rangle$ does not go wrong in the abstract semantics of Figure 1. This is because $\text{lock}_2()$ is rendered unreachable from state $\langle s_2, \texttt{s=U} \rangle$ in the abstract semantics by the $\texttt{assume}(\mathit{false})$ statement as a result of which the model checker does not even analyze $\text{lock}_2()$. However, the type system must type check all code, including code that is *dead*. In particular, it must assign a type to $\text{lock}_2()$. It uses the type $\top$ for this purpose. Then, a type derivation for s_2 illustrating that $\langle s_2, \texttt{s=U} \rangle$ is well-typed is as follows:

$$\dfrac{\dfrac{\text{lock}_1() \ : \ \texttt{s=U} \rightarrow \texttt{s=L} \qquad \texttt{assume}(\mathit{false}) \ : \ \texttt{s=L} \rightarrow \bot}{\text{lock}_1(); \ \texttt{assume}(\mathit{false}) \ : \ \texttt{s=U} \rightarrow \bot} \qquad \text{lock}_2() \ : \ \top}{s_2 \ : \ \texttt{s=U} \rightarrow \bot}$$

Example 3. Consider the following program:

$$s_3 \triangleq \{ \texttt{while} \ (*) \ \texttt{do} \ \{ \texttt{assume}(i \neq 2); i := i + 1 \} \}; \texttt{assume}(i = 2)$$

Suppose the abstraction scheme is predicate abstraction and suppose $\Omega = \{\mathsf{i}{=}0, \mathsf{i}{=}1, \mathsf{i}{=}2\}$. Then, each of states $\langle s_3, \mathsf{i}{=}0\rangle$, $\langle s_3, \mathsf{i}{=}1\rangle$, and $\langle s_3, \mathsf{i}{=}2\rangle$ does not go wrong in our abstract semantics and, likewise, each of them is well-typed in our type system since s_3 has type $\mathsf{i}{=}0 \to \mathsf{i}{=}2 \ \wedge\ \mathsf{i}{=}1 \to \mathsf{i}{=}2 \ \wedge\ \mathsf{i}{=}2 \to \mathsf{i}{=}2$. For instance, a type derivation for s_3 illustrating that state $\langle s_3, \mathsf{i}{=}0\rangle$ is well-typed is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{assume}(i \neq 2) \ :\ \mathsf{i}{=}0 \to \mathsf{i}{=}0 \ \wedge\ \mathsf{i}{=}1 \to \mathsf{i}{=}1 \ \wedge\ \mathsf{i}{=}2 \to \bot \qquad i := i+1 \ :\ \mathsf{i}{=}0 \to \mathsf{i}{=}1 \ \wedge\ \mathsf{i}{=}1 \to \mathsf{i}{=}2}{\texttt{assume}(i \neq 2);\ i := i+1 \ :\ \mathsf{i}{=}0 \to \mathsf{i}{=}1 \ \wedge\ \mathsf{i}{=}1 \to \mathsf{i}{=}2 \ \wedge\ \mathsf{i}{=}2 \to \bot} \qquad \texttt{assume}(i = 2) \ :\ \mathsf{i}{=}0 \to \bot \ \wedge\ \mathsf{i}{=}1 \to \bot \ \wedge\ \mathsf{i}{=}2 \to \mathsf{i}{=}2}{\texttt{while}\ (*)\ \texttt{do}\ \{\ \texttt{assume}(i \neq 2);\ i := i+1\ \}\ :\ \mathsf{i}{=}0 \to (\mathsf{i}{=}0 \ \vee\ \mathsf{i}{=}1 \ \vee\ \mathsf{i}{=}2)}}{s_3 \ :\ \mathsf{i}{=}0 \to \mathsf{i}{=}2}$$

Thus, both the model checker and the type system accept each of states $\langle s_3, \mathsf{i}{=}0\rangle$, $\langle s_3, \mathsf{i}{=}1\rangle$, and $\langle s_3, \mathsf{i}{=}2\rangle$.

6 Related Work

In recent years, there has been a significant surge of interest in type systems for checking temporal safety properties of imperative programs [47, 13, 17, 23, 26]. For instance, consider program s_3 in Example 3 above which has the type $\mathsf{i}{=}0 \to \mathsf{i}{=}2 \ \wedge\ \mathsf{i}{=}1 \to \mathsf{i}{=}2 \ \wedge\ \mathsf{i}{=}2 \to \mathsf{i}{=}2$ in our type system instantiated with the set of abstract contexts $\Omega = \{\mathsf{i}{=}0, \mathsf{i}{=}1, \mathsf{i}{=}2\}$. In CQual [17], which supports references and therefore has a more specialized type system than ours, s_3 would be annotated with a constrained polymorphic type:

$$s_3 : \forall c, c'.\ (ref(l), [l \mapsto int(c)]) \to (ref(l), [l \mapsto int(c')])\ /\ \{(c = 0 \Rightarrow c' = 2), (c = 1 \Rightarrow c' = 2), (c = 2 \Rightarrow c' = 2)\}$$

where $ref(l)$ is a singleton reference type, namely, the type of a reference to the location l, and $int(c)$ is a singleton integer type, namely, the type of the integer constant c. Singleton types are not unusual and have also been used in the type systems of languages such as Xanadu [47] and Vault [13] as well as in the type systems of alias types [45] and refinement types [26].

There is a large body of work on bridging different approaches to static analysis, most notably (i) on relating type systems and control-flow analysis for higher-order functional languages, and (ii) on relating data-flow analysis and model checking for first-order imperative languages.

Type Systems and Control-Flow Analysis. The Amadio-Cardelli type system [2] with recursive types and subtyping has been shown to be equivalent to a certain 0-CFA-based safety analysis by Palsberg and O'Keefe [36] and to a certain form of constrained types by Palsberg and Smith [37], thereby unifying three different views of typing. Heintze [20] proves that four restrictions of 0-CFA are equivalent

to four type systems parameterized by recursive types and subtyping. Palsberg shows that equality-based 0-CFA is equivalent to a type system with recursive types and an unusual notion of subtyping [34]. Palsberg and Pavlopoulou [33] and Amtoft and Turbak [3] show that a class of finitary polyvariant control-flow analyses is equivalent to a type system with finitary polymorphism in the form of union and intersection types. Mossin [28] presents a sound and complete type-based flow analysis in that it predicts a redex if and only if there exists a reduction sequence such that the redex will be reduced. Mossin's approach uses intersection types annotated with flow information; a related approach to flow analysis has been presented by Banerjee [5].

Data-Flow Analysis and Model Checking. Schmidt and Steffen [42, 41, 40] relate data-flow analysis and model checking for first-order imperative languages. They show that the information computed by classical iterative data-flow analyses is the same as that obtained by model checking certain modal mu-calculus formulae on the program's trace-based abstract interpretation (*a.i.*), an operational-semantics-based representation of the program's *a.i.* as a computation tree of traces.

7 Conclusions

We have presented a type system that is equivalent to a model checker for verifying temporal safety properties of imperative programs. Our result highlights the essence of the relationship between type systems and model checking, provides a methodology for studying their relative expressiveness, is a step towards sharing results between them, and motivates synergistic program analyses that can gain the advantages of both approaches without suffering the drawbacks of either.

Two limitations of our current work are that our language lacks features such as higher-order functions, objects, and concurrency, and the type information extracted from the model constructed by our model checker may not be suitable for human reasoning. We intend to explore these issues in the context of specific verification problems. For instance, see [1] for an approach that infers lock types from executions of multithreaded Java programs in the context of verifying race-freedom.

Acknowledgments

We originally proved our equivalence result in the setting of the deadline analysis problem for the interrupt calculus. That result can be found in the first author's Master's thesis [29]. We thank the many people who suggested that we prove the result in a more conventional setting such as the one in this paper. The proof technique remains essentially the same. We would also like to thank Alex Aiken and Jakob Rehof for useful discussions and the anonymous reviewers for insightful comments. We were supported by a National Science Foundation ITR Award number 0112628.

References

1. R. Agarwal and S. D. Stoller. Type inference for parameterized race-free java. In *Proceedings of VMCAI'04*, pages 149–160, January 2004.
2. Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM TOPLAS*, 15(4):575–631, 1993.
3. Torben Amtoft and Franklyn Turbak. Faithful translations between polyvariant flows and polymorphic types. In *Proceedings of ESOP'00*, pages 26–40, 2000.
4. Thomas Ball and Sriram Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of POPL'02*, pages 1–3, 2002.
5. Anindya Banerjee. A modular, polyvariant and type-based closure analysis. In *Proceedings of ICFP'97*, pages 1–10, 1997.
6. M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM LOPLAS*, 2(1-4):17–30, 1993.
7. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proceedings of ICSE'03*, pages 385–395, May 2003.
8. Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: Model checking message-passing programs. In *Proceedings of POPL'02*, pages 45–57, 2002.
9. Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Proceedings of ICFP'01*, pages 193–204, 2001.
10. Patrick Cousot. Types as abstract interpretations. In *Proceedings of POPL'97*, pages 316–331, 1997.
11. Patrick Cousot and Radhia Cousot. Temporal abstract interpretation. In *Proceedings of POPL'00*, pages 12–25, 2000.
12. M. Debbabi, A. Benzakour, and K. Ktari. A synergy between model-checking and type inference for the verification of value-passing higher-order processes. In *Proceedings of AMAST'98*, pages 214–230, 1999.
13. Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of PLDI'01*, pages 59–69, 2001.
14. Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.
15. Cormac Flanagan and Stephen N. Freund. Type inference against races. In *Proceedings of SAS'04*, 2004.
16. Cormac Flanagan, Stephen N. Freund, and Marina Lifshin. Type inference for atomicity. In *Proceedings of TLDI'05*, January 2005.
17. Jeffrey S. Foster, Tachio Terauchi, and Alexander Aiken. Flow-sensitive type qualifiers. In *Proceedings of PLDI'02*, pages 1–12, 2002.
18. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proceedings of CAV'97*, pages 72–83, 1997.
19. Christian Haack and Joe B. Wells. Type error slicing in implicitly typed higher-order languages. In *Proceedings of ESOP'03*, pages 284–301, 2003.
20. Nevin Heintze. Control-flow analysis and type systems. In *Proceedings of SAS'95*, pages 189–206, September 1995.
21. T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Proceedings of CAV'02*, pages 526–538, 2002.
22. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *Proceedings of SPIN'03*, pages 235–239, 2003.
23. Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Proceedings of POPL'02*, pages 331–342, 2002.

24. G. F. Johnson and J. A. Walz. A maximum flow approach to anomaly isolation in unification-based incremental type inference. In *Proceedings of POPL'86*, pages 44–57, 1986.
25. Di Ma. *Bounding the Stack Size of Interrupt-Driven Programs*. PhD thesis, Purdue University, 2004.
26. Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Proceedings of ICFP'03*, 2003.
27. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
28. C. Mossin. Exact flow analysis. In *Proceedings of SAS'97*, pages 250–264, 1997.
29. Mayur Naik. A type system equivalent to a model checker. Master's thesis, Purdue University, 2004.
30. Mayur Naik and Jens Palsberg. A type system equivalent to a model checker. In *Proceedings of ESOP'05*, 2005. Full version with proofs available at `http://www.cs.stanford.edu/~mhn/pubs/esop05.html`.
31. K. S. Namjoshi. Certifying model checkers. In *Proceedings of CAV'01*, pages 2–12, 2001.
32. K. S. Namjoshi. Lifting temporal proofs through abstractions. In *Proceedings of VMCAI'03*, pages 174–188, 2003.
33. J. Palsberg and C. Pavlopoulou. From polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11(3):263–317, May 2001.
34. Jens Palsberg. Equality-based flow analysis versus recursive types. *ACM TOPLAS*, 20(6):1251–1264, 1998.
35. Jens Palsberg and Di Ma. A typed interrupt calculus. In *Proceedings of FTRTFT'02*, pages 291–310, September 2002.
36. Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. *ACM TOPLAS*, 17(4):576–599, July 1995.
37. Jens Palsberg and Scott Smith. Constrained types and their expressiveness. *ACM TOPLAS*, 18(5):519–527, September 1996.
38. Doron Peled, Amir Pnueli, and Lenore D. Zuck. From falsification to verification. In *Proceedings of FSTTCS'01*, pages 292–304, 2001.
39. Doron Peled and Lenore D. Zuck. From model checking to a temporal proof. In *Proceedings of SPIN'01*, pages 1–14, 2001.
40. David Schmidt and Bernhard Steffen. Program analysis as model checking of abstract interpretations. In *Proceedings of SAS'98*, pages 351–380, 1998.
41. David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of POPL'98*, pages 38–48, 1998.
42. Bernhard Steffen. Data flow analysis as model checking. In *Proceedings of TACS'91, Theoretical Aspects of Computer Science*, pages 346–364, 1991.
43. Li Tan and Rance Cleaveland. Evidence-based model checking. In *Proceedings of CAV'02*, pages 455–470, 2002.
44. Frank Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM TOSEM*, 10(1):5–55, 2001.
45. David Walker and Greg Morrisett. Alias types for recursive data structures. In *Proceedings of TIC'00*, pages 177–206, 2001.
46. Mitchell Wand. Finding the source of type errors. In *Proceedings of POPL'86*, pages 38–43, 1986.
47. Hongwei Xi. Imperative programming with dependent types. In *Proceedings of LICS'00*, pages 375–387, 2000.

Appendix

Lemma 6. (Progress) *If $\langle s, \omega_m \rangle$ is well-typed then $\langle s, \omega_m \rangle$ is not stuck.*

Proof. By induction on the structure of s. There are 6 cases depending upon the form of s. (In cases (1), (2), (5), and (6), we do not use the hypothesis that $\langle s, \omega_m \rangle$ is well-typed.)

1. $s = p$. Immediate from rule (1) and the fact that $\forall i \in \Omega : \delta_p(i) \neq \emptyset$.
2. $s =$ `assume`(e). Immediate from rules (2) and (3).
3. $s =$ `assert`(e). From $\langle s, \omega_m \rangle$ is well-typed and rule (12), we have $s : \bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j)$ and $m \in A$. From $s : \bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j)$ and rule (15), we have $A \subseteq \delta_e$. From $m \in A$ and $A \subseteq \delta_e$, we have $m \in \delta_e$. From $m \in \delta_e$ and rule (4), we have $\langle s, \omega_m \rangle \hookrightarrow \omega_m$, whence $\langle s, \omega_m \rangle$ is not stuck.
4. $s = s_1;\ s_2$. From $\langle s, \omega_m \rangle$ is well-typed and rule (12), we have $s : \bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j)$ and $m \in A$. From $s : \bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j)$ and rule (16), we have $s_1 : \bigwedge_{i \in A'}(\omega_i \rightarrow \bigvee_{j \in B'_i} \omega_j)$ and $A \subseteq A'$. From $m \in A$ and $A \subseteq A'$, we have $m \in A'$. From $s_1 : \bigwedge_{i \in A'}(\omega_i \rightarrow \bigvee_{j \in B'_i} \omega_j)$ and $m \in A'$ and rule (12), we have $\langle s_1, \omega_m \rangle$ is well-typed. From $\langle s_1, \omega_m \rangle$ is well-typed and the induction hypothesis, we have $\langle s_1, \omega_m \rangle$ is not stuck. From $\langle s_1, \omega_m \rangle$ is not stuck, we have $\exists a :\ \langle s_1, \omega_m \rangle \hookrightarrow a$. There are 3 cases depending upon the form of a. In each case, we will prove that $\langle s, \omega_m \rangle$ is not stuck.
   - $a = \omega'$. From rule (5), we have $\langle s, \omega_m \rangle \hookrightarrow \langle s_2, \omega' \rangle$.
   - $a =$ error. From rule (6), we have $\langle s, \omega_m \rangle \hookrightarrow$ error.
   - $a = \langle s'_1, \omega' \rangle$. From rule (7), we have $\langle s, \omega_m \rangle \hookrightarrow \langle s'_1; s_2, \omega' \rangle$.
5. $s =$ `if` $(*)$ `then` s_1 `else` s_2. Immediate from either of rules (8) and (9).
6. $s =$ `while` $(*)$ `do` s'. Immediate from either of rules (10) and (11).

Lemma 7. *If $s : \bigwedge_{i \in C}(\omega_i \rightarrow \bigvee_{j \in D_i} \omega_j)$ and $m \in C$ and $\langle s, \omega_m \rangle \hookrightarrow \langle s', \omega_n \rangle$ then $s' : \bigwedge_{i \in E}(\omega_i \rightarrow \bigvee_{j \in F_i} \omega_j)$ and $n \in E$ and $F_n \subseteq D_m$.*

Proof. See technical report [30].

Lemma 8. (Single-step Type Preservation) *If $\langle s, \omega_m \rangle$ is well-typed and $\langle s, \omega_m \rangle \hookrightarrow \langle s', \omega_n \rangle$ then $\langle s', \omega_n \rangle$ is well-typed.*

Proof. From $\langle s, \omega_m \rangle$ is well-typed and rule (12), we have $s : \bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j)$ and $m \in A$. From $s : \bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j)$ and $m \in A$ and $\langle s, \omega_m \rangle \hookrightarrow \langle s', \omega_n \rangle$ and lemma (7), we have $s' : \bigwedge_{i \in A'}(\omega_i \rightarrow \bigvee_{j \in B'_i} \omega_j)$ and $n \in A'$. From $s' : \bigwedge_{i \in A'}(\omega_i \rightarrow \bigvee_{j \in B'_i} \omega_j)$ and $n \in A'$ and rule (12), we have $\langle s', \omega_n \rangle$ is well-typed.

Lemma 9. (Multi-step Type Preservation) *If $\langle s, \omega_m \rangle$ is well-typed and $\langle s, \omega_m \rangle \hookrightarrow^t \langle s', \omega_n \rangle$ then $\langle s', \omega_n \rangle$ is well-typed.*

Proof. By induction on t (using lemma (8)).

Lemma 10. *We have:*

$$\mathbb{A}^s \subseteq \begin{cases} \mathbb{A}^{s_1} & \text{if } s = s_1;\ s_2 \\ \mathbb{A}^{s_1} \cap \mathbb{A}^{s_2} & \text{if } s = \texttt{if } (*) \texttt{ then } s_1 \texttt{ else } s_2 \\ \mathbb{A}^{s'} & \text{if } s = \texttt{while } (*) \texttt{ do } s' \end{cases}$$

Proof. See technical report [30].

Lemma 11. *If* $s = s_1;\ s_2$ *then* $\bigcup_{i \in \mathbb{A}^s} \mathbb{B}^{s_1,i} \subseteq \mathbb{A}^{s_2}$. *If* $s = \texttt{while } (*) \texttt{ do } s'$ *then* $\bigcup_{i \in \mathbb{A}^s} \mathbb{B}^{s',i} \subseteq \mathbb{A}^s$.

Proof. See technical report [30].

Lemma 12. (Typability) $s : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.

Proof. By induction on the structure of s. There are 6 cases depending upon the form of s:

- $s = p$. From defn. (1) and rule (1), we have $\mathbb{A}^s = \Omega$. From defn. (2), we have $\forall i \in \Omega :\ \mathbb{B}^{s,i} = \delta_p(i)$. From $\mathbb{A}^s = \Omega$ and $\forall i \in \mathbb{A}^s :\ \mathbb{B}^{s,i} = \delta_p(i)$ and rule (13), we have $s : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.
- $s = \texttt{assume}(e)$. From defn. (1) and rules (2) and (3), we have $\mathbb{A}^s = \Omega$. From defn. (2), we have $\forall i \in \delta_e :\ \mathbb{B}^{s,i} = \{i\}$ and $\forall i \notin \delta_e :\ \mathbb{B}^{s,i} = \emptyset$. From $\mathbb{A}^s = \Omega$ and $\forall i \in \delta_e :\ \mathbb{B}^{s,i} = \{i\}$ and $\forall i \notin \delta_e :\ \mathbb{B}^{s,i} = \emptyset$ and rule (14), we have $s : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.
- $s = \texttt{assert}(e)$. From defn. (1) and rule (4), we have $\mathbb{A}^s = \delta_e$. From defn. (2), we have $\forall i \in \delta_e : \mathbb{B}^{s,i} = \{i\}$. From $\mathbb{A}^s = \delta_e$ and $\forall i \in \mathbb{A}^s :\ \mathbb{B}^{s,i} = \{i\}$ and rule (15), we have $s : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.
- $s = s_1;\ s_2$. From the induction hypothesis, we have $s_1 : \bigwedge_{i \in \mathbb{A}^{s_1}} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s_1,i}} \omega_j)$. and $s_2 : \bigwedge_{i \in \mathbb{A}^{s_2}} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s_2,i}} \omega_j)$. From lemma (10), we have $\mathbb{A}^s \subseteq \mathbb{A}^{s_1}$. From lemma (11), we have $\bigcup_{i \in \mathbb{A}^s} \mathbb{B}^{s_1,i} \subseteq \mathbb{A}^{s_2}$. From defn. (2), we have $\mathbb{B}^{s,i} = \bigcup \{\, \mathbb{B}^{s_2,j} \mid j \in \mathbb{B}^{s_1,i} \,\}$. From $s_1 : \bigwedge_{i \in \mathbb{A}^{s_1}} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s_1,i}} \omega_j)$ and $s_2 : \bigwedge_{i \in \mathbb{A}^{s_2}} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s_2,i}} \omega_j)$ and $\mathbb{A}^s \subseteq \mathbb{A}^{s_1}$ and $\bigcup_{i \in \mathbb{A}^s} \mathbb{B}^{s_1,i} \subseteq \mathbb{A}^{s_2}$ and $\mathbb{B}^{s,i} = \bigcup \{ \mathbb{B}^{s_2,j} \mid j \in \mathbb{B}^{s_1,i} \}$ and rule (16), we have $s : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.
- $s = \texttt{if } (*) \texttt{ then } s_1 \texttt{ else } s_2$. From the induction hypothesis, we have $s_1 : \bigwedge_{i \in \mathbb{A}^{s_1}} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s_1,i}} \omega_j)$ and $s_2 : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s_2,i}} \omega_j)$. From lemma (10), we have $\mathbb{A}^s \subseteq \mathbb{A}^{s_1}$ and $\mathbb{A}^s \subseteq \mathbb{A}^{s_2}$. From defn. (2), we have $\mathbb{B}^{s,i} = \mathbb{B}^{s_1,i} \cup \mathbb{B}^{s_2,i}$. From $s_1 : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s_1,i}} \omega_j)$ and $s_2 : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s_2,i}} \omega_j)$ and $\mathbb{A}^s \subseteq \mathbb{A}^{s_1}$ and $\mathbb{A}^s \subseteq \mathbb{A}^{s_2}$ and $\mathbb{B}^{s,i} = \mathbb{B}^{s_1,i} \cup \mathbb{B}^{s_2,i}$ and rule (17), we have $s : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.
- $s = \texttt{while } (*) \texttt{ do } s'$. From the induction hypothesis, we have $s' : \bigwedge_{i \in \mathbb{A}^{s'}} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s',i}} \omega_j)$. From lemma (10), we have $\mathbb{A}^s \subseteq \mathbb{A}^{s'}$. From lemma (11), we have $\bigcup_{i \in \mathbb{A}^s} \mathbb{B}^{s',i} \subseteq \mathbb{A}^s$. From defn. (2), we have $\mathbb{B}^{s,i} = \mu X.(\{i\} \cup \{\mathbb{B}^{s',j} \mid j \in X\})$. From $s' : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s',i}} \omega_j)$ and $\mathbb{A}^s \subseteq \mathbb{A}^{s'}$ and $\bigcup_{i \in \mathbb{A}^s} \mathbb{B}^{s',i} \subseteq \mathbb{A}^s$ and $\mathbb{B}^{s,i} = \mu X.(\{i\} \cup \{\mathbb{B}^{s',j} \mid j \in X\})$ and rule (18), we have $s : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.

Instant Polymorphic Type Systems for Mobile Process Calculi: Just Add Reduction Rules and Close*

Henning Makholm and J.B. Wells

Heriot-Watt University

Abstract. Many different *mobile process calculi* have been invented, and for each some number of type systems has been developed. Soundness and other properties must be proved separately for each calculus and type system. We present the *generic* polymorphic type system Poly✶ which works for a wide range of mobile process calculi, including the π-calculus and Mobile Ambients. For any calculus satisfying some general syntactic conditions, well-formedness rules for types are derived automatically from the reduction rules and Poly✶ works otherwise unchanged. The derived type system is automatically sound (i.e., has subject reduction) and often more precise than previous type systems for the calculus, due to Poly✶'s *spatial polymorphism*. We present an implemented type inference algorithm for Poly✶ which automatically constructs a typing given a set of reduction rules and a term to be typed. The generated typings are *principal* with respect to certain natural type shape constraints.

1 Introduction

Many calculi that intend to capture the essence of *mobile* and *distributed* computing have been invented. The most well-known of these are probably the π-calculus [18] and the *Mobile Ambients* calculus (MA) by Cardelli and Gordon [8], but they haveinspired the subsequent development of a wide variety of variants and alternatives, which are variously argued to be easier to program in or reason about, and/or closer to some operational intuition about how programs in a mobile, distributed setting can be implemented. The field stays productive; new calculi are still being proposed and there is not a clear consensus about what should be considered *the* fundamental mobility calculus.

The majority of these calculi share the basic architecture of MA: They borrow from the π-calculus the syntactic machinery for talking about sets of parallel, communicating processes, plus its primitive operator ν for generating unique names. To this they add some kind of *spatial structure*, usually in the form of a tree of locations where processes can reside. The tree can generally evolve under program control as the processes in it execute; the different calculi provide quite different primitives for mutating it. Mobility calculi also provide for *communication* between processes that are near each other, usually modelled on the communication primitive of the π-calculus, but again with variations and often extended with the possibility to communicate "capabilities", "paths", or other restricted pieces of process syntax, rather than just names.

* Supported by EC FP5/IST/FET grant IST-2001-33477 "DART", and Sun Microsystems equipment grant EDUD-7826-990410-US.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 389–407, 2005.

Most process calculi have an associated *type system*, either one that was designed with the calculus from the beginning, or one that was retrofitted later. These type systems are closely tied to a specific calculus and its particular primitives. Once a type system has been designed and its properties (such as soundness or the applicability of a particular type inference algorithm) have been proved, it is in general not trivial to see whether these properties will survive changes to the calculus.

1.1 A Generic Type System

In contrast, this paper presents the *generic* type system Poly✶ which works for a wide range of mobile process calculi. To use Poly✶, one simply instantiates it with the reduction rules that specify the semantics of the target calculus's primitives. From this, a set of provably sound well-formedness rules for types can be *mechanically* produced, guaranteeing that types that satisfy the rules are sound with respect to the reduction rules, i.e., subject reduction holds. The reduction rules can also be used to guide an automatic *type inference* algorithm for the instantiated type system. The inference algorithm produces a type which is *principal* with respect to certain natural constraints on the shape of types. Our implementation offers several possibilities for tuning the *precision* of the type system it implements, but the use of these is optional — it will always produce a typing even when given only the raw reduction rules of the target calculus.

For this to work, the target calculus must make one small concession to Poly✶, namely that its *syntax* is sufficiently regular that the implementation can make sense of its terms and reduction rules. We define a **metacalculus** Meta✶ which gives a syntax that is easy to parse and manipulate, while flexible enough that many calculi can be viewed as instances of it without deviating much from their native notations. Meta✶ does not include any fixed *semantics* except for the usual semantics of parallelism and name restriction, but instead provides a common notion of substitution and a notation for rewriting rules that fits how semantics for process calculi are usually defined.

1.2 Poly✶'s Relation to Other Reasoning Principles

A long-term goal of Poly✶ is to make it possible to view many previously existing mobility calculi type systems as instances of Poly✶, at least with regards to using the type system to statically verify that certain bad behaviours do not occur. The design we present here does not quite reach that point; there are features of existing type systems that we have not yet incorporated in Poly✶. We believe it will be particularly important to express some form of the *single-threaded* locations introduced by the original type system for Safe Ambients [16].

We do not expect actual programming environments based on mobility calculi to use the fully general Poly✶ formalism as their type discipline. Considerations of performance and integration will generally dictate that production environments instead use hand-crafted specialised type systems for the language they support, though *ideas* from Poly✶ may well be employed.

A generic implementation of Poly✶, such as the one we present here, should be a valuable tool for *exploring the design space* for mobility calculi in general. It will make it easy to change some aspect of one's rewriting rules, try to analyse some terms, and see which effect the new rules have on, for example, the interference-control properties

of one's calculus. At the same time, our Poly✶ implementation makes it easy to experiment with exactly how strong a type system one wants to use in practice, because our implementation supports tuning the precision of types in very small steps.[1]

Like every nontrivial type system with an inference algorithm, Poly✶ can be used as a *control/data flow analysis* to provide the substratum for more specialised automatic *program analyses*.[2] (Readers who are uncomfortable about applying the term "type system" to Poly✶ are invited to think "versatile program analysis framework" each time we write "type system".) However, we have no pretension of subsuming all other analysis techniques for mobility or process calculi in general. Process calculi have provided the setting for many advanced techniques for reasoning about, for example, behavioural equivalence of processes. Poly✶ does not claim to compete with these.

1.3 Spatial Polymorphism

The Poly✶ type system descends from (but significantly generalises and enhances) our earlier work [2] on PolyA, a polymorphic type system specific to Mobile Ambients. It inherits from PolyA the difference from most other type systems for mobility calculi that the emphasis is on types for *processes* rather than types for (ambient or channel) *names*.[3] In fact, types for names have completely vanished: A name has no intrinsic type of its own, but is distinguished solely by the way it can be used to form processes.

Poly✶ works by approximating the set of terms a given term can possibly evolve to using the given reduction rules. Its central concept is that of a **shape predicate** which is an automaton that describes a set of process terms. Shape predicates that satisfy certain well-formedness rules are **types**. These rules are derived from the reduction rules of the target calculus and guarantee that the set of terms denoted by a type is closed under the reduction relation, i.e., *subject reduction* holds.

This design gives rise to a new (with PolyA) form of polymorphism that we call **spatial polymorphism**. The type of a process may depend on where in the spatial structure it is found. When the process moves, it may come under influence of another part of the type which allows more reductions. For example, consider a calculus which has the single reduction rule $\mathtt{a[eat\ b \mid P] \mid b[Q]} \hookrightarrow \mathtt{a[P \mid b[Q]]}$. In this calculus, the term $\mathtt{x[eat\ z1 \mid eat\ z2] \mid y1[eat\ x \mid z1[0]] \mid y2[eat\ x \mid z2[0]]}$ has a Poly✶ type, shown in Figure 1, that says that x[] may contain z1[] when it it inside y1[], or z2[] when it it inside y2[], but can contain neither when it is found at the top level of the term. Thus Poly✶ can prove that the term satisfies the safety policy that z1 and z2 may never be found side by side. To our knowledge, type systems based on earlier paradigms cannot do this.

With spatial polymorphism, *movement* is what triggers the generation of a polymorphic variant of the original analysis for a piece of code. This is different from, and

[1] These fine tuning options are omitted from this paper due to lack of space, but they are described in detail in the implementation's documentation.

[2] Indeed it is well known [20, 3] that the difference between an advanced flow analysis and an advanced type system is often just a question of different perspectives on the same underlying machinery. The presentation of Poly✶ is closer to the data-flow viewpoint than is common for type systems, though this of course does not make Poly✶ any less a type system.

[3] There are a number of type systems for process calculi *without* an explicit notion of locations which assign types to processes rather than names, for example [4, 14, 27, 11].

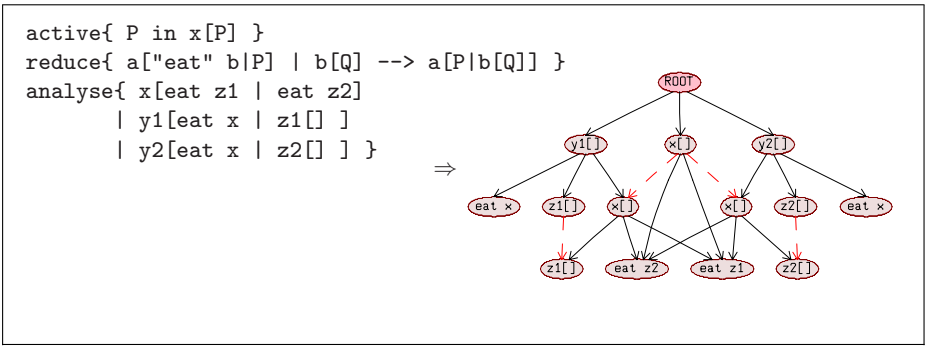


Fig. 1. Input to our type inference implementation for analysing a term in the fictional "eat calculus", and the inferred type graph as rendered by the VCG graph layout tool [23] (the dashed lines represent subtyping edges)

orthogonal to, the more conventional form of name-parametric polymorphism in the polymorphic π-calculus [24], where it is *communication* that causes polymorphic variants to be created. Poly✶ does not support the latter form of polymorphism (and neither does any type system for a mobility calculus with explicit locations that we are aware of); we leave it to future work to try to combine the strengths of these two principles.

1.4 Notation and Preliminaries

$\boxed{X}$, where X is any metavariable symbol, stands for the set that X ranges over. $\mathcal{P}_{\text{fin}}(A)$ is the set of finite subsets of the set A. $A \xrightarrow{\text{fin}} B$ is the set of finite partial maps from A to B. Dom f is the set of x's such that $f(x)$ is defined. In contexts where a sequence of similar objects are indexed with indexes up to k, it is to be understood that k can be any integer ≥ 0. Thus, if the first index is 0, the sequence must have at least one element; sequences indexed from 1 to k may be empty.

2 Meta✶: A Metacalculus of Concurrent Processes

The metacalculus Meta✶ defined in this section is the *syntactic* setting for Poly✶. Its role is to let us present the generic properties of Poly✶ without resorting to handwaving. Though we define a reduction relation and some other formal properties for Meta✶, these exist solely as support for making formal statements about Poly✶. We do not intend Meta✶ to take the place of any existing calculi or frameworks.

As a first approximation, Meta✶ is a "syntax without a semantics" except that it does give semantics to a few basic constructs, e.g., process replication and substitution.

2.1 Terms

Figure 2 shows the syntax of process terms in Meta✶. The trivial process `0`, parallel composition of processes $P \mid Q$, process replication $!P$, and name restriction $\nu(x).P$ are all well-known from most process calculi, including π-calculus and MA. They are given their usual behaviour by the structural congruence relation $\equiv$.

Names: $x,y ::= \mathsf{a} \mid \mathsf{b} \ldots \mathsf{z} \mid \mathsf{aa} \mid \mathsf{ab} \ldots \mathsf{eas} \mid \mathsf{eat} \mid \mathsf{eau} \ldots \mid \texttt{[]} \mid \texttt{\^{}} \mid \texttt{:} \mid \texttt{*} \mid \texttt{/} \ldots \mid \bullet$
Sub-forms: $f ::= x_0\, x_1 \ldots x_k$
Messages: $M,N ::= f \mid \mathsf{0} \mid M.N$
Elements: $E ::= x \mid (x_1,x_1,\ldots,x_k) \mid \texttt{<}M_1,M_2,\ldots,M_k\texttt{>}$
Forms: $F ::= E_0\, E_1 \ldots E_k$
Processes: $P,Q,R ::= F.P \mid \,!P \mid \nu(x).P \mid \mathsf{0} \mid (P \mid Q)$

Free and bound names in terms are defined thus (the omitted cases being purely structural):

$$\mathrm{FN}(x) = \{x\} \qquad \mathrm{BN}(x) = \varnothing$$
$$\mathrm{FN}((x_1,\ldots,x_k)) = \varnothing \qquad \mathrm{BN}((x_1,\ldots,x_k)) = \{x_1,\ldots,x_k\}$$
$$\mathrm{FN}(F.P) = \mathrm{FN}(F) \cup (\mathrm{FN}(P) \setminus \mathrm{BN}(F)) \qquad \mathrm{BN}(F.P) = \mathrm{BN}(F) \cup \mathrm{BN}(P)$$
$$\mathrm{FN}(\nu(x).P) = \mathrm{FN}(P) \setminus \{x\} \qquad \mathrm{BN}(\nu(x).P) = \mathrm{BN}(P)$$

$$P \equiv P \qquad P \equiv Q \Longrightarrow Q \equiv P \qquad P \equiv Q \wedge Q \equiv R \Longrightarrow P \equiv R \qquad P \mid Q \equiv Q \mid P$$
$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad P \mid \mathsf{0} \equiv P \qquad !P \equiv P \mid !P \qquad !\mathsf{0} \equiv \mathsf{0}$$
$$P \equiv Q \Longrightarrow F.P \equiv F.Q \qquad P \equiv Q \Longrightarrow !P \equiv !Q \qquad P \equiv Q \Longrightarrow \nu(x).P \equiv \nu(x).Q$$
$$P \equiv Q \Longrightarrow P \mid R \equiv Q \mid R \qquad x \notin \mathrm{FN}(F) \wedge x \notin \mathrm{BN}(F) \Longrightarrow F.\nu(x).P \equiv \nu(x).F.P$$
$$x \notin \mathrm{FN}(P) \Longrightarrow P \mid \nu(x).Q \equiv \nu(x).(P \mid Q)$$
$$y \notin \mathrm{FN}(P) \Longrightarrow \nu(x).P \equiv \nu(y).[x := y]P \qquad \nu(x).\nu(y).P \equiv \nu(y).\nu(x).P$$

Fig. 2. Syntax of Meta✶ plus its structural congruence relation

Meta✶ amalgamates all other process constructors into the general concept of a **form**. Forms have no intrinsic meaning until a set of reduction rules give them one. Examples of forms include the communication actions "$x\texttt{<}y\texttt{>}$" and "$x\,(y)$" from the π-calculus, the movement capabilities "in x", "out x", and "open x" from Mobile Ambients, and even ambient boundaries themselves, which we write as "$x\,\texttt{[]}$". We support the traditional syntax "$x[P]$" for ambients by interpreting "$E_1 \ldots E_k[P]E'_1 \ldots E'_{k'}$" as syntactic sugar for "$E_1 \ldots E_k\,\texttt{[]}\,E'_1 \ldots E'_k.P$". Except for this syntactic convention, the symbol `[]` has no special interpretation in Meta✶ and it is a (single) name just like in and out. The process $F.\mathsf{0}$ can be abbreviated as F.

A form consists of a nonempty sequence of **elements**, each of which is either a **name**, a **binding** element, or a **message** element. Names are used to name channels, ambients, and so on, but also work as keywords that distinguish forms with different roles in the calculus. A keyword is simply a free name that is matched explicitly by some reduction rule. Most non-alphanumeric ASCII characters that do not have any special meaning (`^`, `:`, `*`, `/`, etc.) are also names and so can be used as keywords. With these we can encode, e.g., annotated communication actions like "$\langle M \rangle^*$" or "$(x)^y$" from Boxed Ambients [5] using pseudo-TEX notation as the forms "`<`M`>^*`" and "`(`x`)^`y".

Binding elements $(x_1,\ldots,x_k)$ are used to create forms that bind names in the process they are applied to. The canonical use of this is for constructing receive actions, but again the meaning of the form is specified only by the reduction rules. Message elements `<`$\cdots$`>` allow a form to contain other forms, which — given appropriate reduction rules for communication — can later be substituted into processes. For technical reasons we have to restrict the forms contained in message elements in that they cannot contain message

or binding elements themselves. We refer to such restricted forms and their elements as **sub-forms** and **sub-elements**. In future work we hope to be able to handle calculi such as the spi-calculus [1] which communicate structured messages.

It is not uncommon for calculi to prefer using an explicit recursion construction "$P ::= \mathbf{rec}\ X.P$" to express infinite behaviour rather than the process replication operator "!". There are certain technical problems with supporting this directly in Meta✶ (which may however be approachable by novel techniques involving regular grammars developed by Nielson et al. [19]). In the common case where the target calculus does not allow location boundaries to come between the **rec** X binder and the bound X, it can easily be simulated in Meta✶ by adding the reduction rule $\mathsf{spawn\ a} \mid \mathsf{rec\ a.P} \hookrightarrow \mathsf{P}$ and then representing **rec** $X.\cdots.X$ as $\nu(\mathsf{x}).(\mathsf{spawn\ x} \mid \mathsf{!rec\ x}.\cdots.\mathsf{spawn\ x})$.

2.2 Well-Scoped Terms

The process term P is **well scoped** iff it contains no nested binding of the same name and none of its free names also appear bound in the term. Formally, it is required that (1) $\mathrm{BN}(P)$ and $\mathrm{FN}(P)$ are disjoint, (2) whenever P contains $F.Q$, $\mathrm{BN}(F)$ and $\mathrm{BN}(Q)$ are disjoint, and (3) whenever P contains $\nu(x).Q$, $x \notin \mathrm{BN}(Q)$.

We generally require that terms are always well scoped. The reduction rules in an instantiation of Meta✶ must preserve well-scopedness. This simplifies the type analysis because then we do not have to support α-conversion of ordinary binding elements.

We must still handle α-conversion of private names, which is built into the $\equiv$ relation, but we will assume that it is not used to create terms that are not well scoped.

2.3 Substitutions

Substitutions in Meta✶ substitute *messages* for *names*. The fact that entire *processes* cannot be substituted is an important technical premise of Poly✶; it means that substitution can preserve well-scopedness. It is remarkable that mobility calculi in general refrain from substituting processes; calculi such as Seal [25] and $\mathbf{M^3}$ [12] which allow exchange of entire processes do it by local *movement* rather than *substitution*. This probably reflects the intuition that a running process is harder to distribute across a recipient process than a mere name or code sequence.

A (term) **substitution** $\mathcal{S}$ is a finite map from names to messages. Figure 3 defines the action of $\mathcal{S}$ on the various syntactic classes of Meta✶. In Mobile Ambients and its descendant calculi, the value exchanged in a communication operation can be either a name or a (sequence of) capabilities. The former is the case in reduction $\langle \mathsf{b} \rangle \mid (a).\mathsf{out}\, a.\mathsf{0} \hookrightarrow \mathsf{out\ b.0}$ and the latter in $\langle \mathsf{in\ b} \rangle \mid (a).\mathsf{x}[a.\mathsf{in\ c.0}] \hookrightarrow \mathsf{x}[\mathsf{in\ b.in\ c.0}]$. To support this, Fig. 3 contains special cases for the syntactic cases $M ::= F$ and $P ::= F.P$ when the form F is a lone name. In that case the substitution for the name is inserted directly into the message (or process structure).

In cases like $\{\mathsf{a} \mapsto \mathsf{b}\}^{\mathbf{P}}\mathsf{x}[\mathsf{out\ a.0}]$ where the substituted name occurs properly inside a form, the substitution is carried out componentwise for each form element, and the name is replaced in the rule for $\mathcal{S}^{\mathbf{E}}x$. In this context the replacement must be a name too. This will be false if the term tries to reduce as $\langle \mathsf{in\ b} \rangle \mid (a).\mathsf{out}\, a.\mathsf{0} \hookrightarrow \mathsf{out}\,(\mathsf{in\ b}).\mathsf{0}$. The published formalisms of most ambient-inspired calculi usually regard "$\mathsf{out}\,(\mathsf{in\ b})$" as *syntactically* possible but *semantically* meaningless. That this configuration cannot occur is often the most basic soundness property of type systems for such calculi.

$$\mathcal{S}^{\mathbf{E}}x = \begin{cases} x & \text{when } x \notin \mathrm{Dom}\,\mathcal{S} \\ y & \text{when } \mathcal{S}(x) = y \text{ for some } y \\ \bullet & \text{otherwise} \end{cases} \qquad \mathcal{S}^{\mathbf{M}}x = \begin{cases} \mathcal{S}(x) & \text{when } x \in \mathrm{Dom}\,\mathcal{S} \\ x & \text{otherwise} \end{cases}$$

$$\mathcal{S}^{\mathbf{P}}(\nu(x).P) = \nu(x).\mathcal{S}^{\mathbf{P}}P \qquad \mathcal{S}^{\mathbf{P}}(x.P) = \begin{cases} \mathcal{S}(x)_*(\mathcal{S}^{\mathbf{P}}P) & \text{when } x \in \mathrm{Dom}\,\mathcal{S} \\ x.(\mathcal{S}^{\mathbf{P}}P) & \text{otherwise} \end{cases}$$

$$(M.N)_*P = M_*(N_*P) \qquad \mathtt{0}_*P = P \qquad f_*P = f.P$$

Fig. 3. The actions of term substitution. $\mathcal{S}^{\mathbf{M}}$ is the action on messages, $\mathcal{S}^{\mathbf{E}}$ the action on elements, $\mathcal{S}^{\mathbf{F}}$ on (sub)forms, and $\mathcal{S}^{\mathbf{P}}$ on processes. The omitted cases (including the one for $\mathcal{S}^{\mathbf{F}}$) simply substitute componentwise into the syntactic element in question. The M_*P helper operator serves to linearise messages once we do not need to keep track of whether they are composite or not. (In other systems, this is often done by the structural congruence relation instead)

In Meta✶ such a semantic error becomes a syntactic one: It is simply not possible to use an entire form as an element (except indirectly through a message element). If, at runtime, a substitution nevertheless tries to do so, we substitute the special name "•", which is to be interpreted as, "an erroneous substitution happened here". Thus, with the MA communication rule, Meta✶ reduces `<in b>.0 | (a).out a.0` $\hookrightarrow$ `out`•`.0`. This convention is technically convenient because it allows us to bound the nesting depth of forms (using the sub-form restriction). Because most published calculi attach no semantics to forms like "out (in b)", we do not lose any real expressiveness.

Forms that contain • are inert in Meta✶ unless there are reduction rules that explicitly match on •. The calculus designer can also define reduction rules that create •'s in other situations to mark reduction results as "erroneous". For example, in the polyadic π-calculus, it is usually considered a run-time error if someone tries to send an m-tuple on a channel where another process is listening for an n-tuple, with $n \neq m$. By writing explicit rules[4] for such situations, they can be handled in parallel with malformed substitutions. (One cannot straightforwardly write patterns to test for malformed substitutions, which is one reason for building the generation of • into Meta✶).

In either case, the Poly✶ type system will conservatively estimate *whether* (and where) a • can occur. Which conclusions to draw from this (e.g., rejecting the input program due to "type error") is up to the designer of the calculus.

The definitions in Figure 3 do not worry about name capture. In general, therefore, $\mathcal{S}^{\mathbf{X}}X$ is only intuitively correct if $\mathrm{BN}(X)$ is disjoint from the names mentioned in $\mathcal{S}$. In practise, this will always follow from the assumption that all terms are well scoped.

2.4 Reduction Rules

Figure 4 defines most of the syntax and semantics of reduction rules for Meta✶. Our full theory (and implementation) allows a slightly more expressive template language

[4] E.g., **reduce**{`<M1,M2>.P | (x1,x2,x3).Q` $\hookrightarrow$ •`.0`} for $(m,n) = (2,3)$. Our implementation provides an extension for writing a single rule that catches all pairs (m,n) at once.

$$
\begin{array}{lll}
\text{Name variables:} & \mathring{x} ::= \mathtt{a} \mid \mathtt{b} \mid \mathtt{c} \mid \cdots & \\
\text{Message variables:} & \mathring{m} ::= \mathtt{M} \mid \mathtt{N} \mid \cdots & \\
\text{Process variables:} & \mathring{p} ::= \mathtt{P} \mid \mathtt{Q} \mid \mathtt{R} \mid \cdots & \\
\text{Substitutes:} & \underline{s} ::= \mathring{x} \mid \mathring{m} & \\
\text{Element templates:} & \underline{E} ::= \mathring{x} \mid x \mid \mathtt{(}\mathring{x}_1,\ldots,\mathring{x}_k\mathtt{)} \mid \mathtt{<}\mathring{m}_1,\ldots,\mathring{m}_k\mathtt{>} & \\
\text{Forms templates:} & \underline{F} ::= \underline{E}_0\,\underline{E}_1\ldots\underline{E}_k & \\
\text{Process templates:} & \underline{P} ::= \mathring{p} \mid \underline{F}.\underline{P} \mid \mathtt{0} \mid (\underline{P}_1 \mid \underline{P}_2) & \\
 & \quad \mid \{\mathring{x}_0 := \underline{s}_0,\ldots,\mathring{x}_k := \underline{s}_k\}\ \mathring{p} & (\mathsf{R}) \\
\text{Rules:} & \mathcal{R}^1 ::= \mathbf{reduce}\{\underline{P}_1 \hookrightarrow \underline{P}_2\} \mid \mathbf{active}\{\mathring{p}\ \mathbf{in}\ \underline{P}\} & \\
\text{Rulesets:} & \mathcal{R} \in \mathcal{P}_{\text{fin}}(\boxed{\mathcal{R}^1}) &
\end{array}
$$

The syntactic choice marked (R) is allowed only in a **reduce** rule to the *right* of the arrow.

Let an **term instantiation** $\mathcal{V}$ map $\boxed{\mathring{x}}$ to $\boxed{x} \setminus \{\bullet\}$, $\boxed{\mathring{m}}$ to $\boxed{M}$, and $\boxed{\mathring{p}}$ to $\boxed{P}$. It applies to templates strictly componentwise, except for the case that fills in and applies a substitution:

$$\mathcal{V}^{\mathbf{P}}(\{\ldots,\mathring{x}_i := \underline{s}_i,\ldots\}\ \mathring{p}) = \{\ldots,\mathcal{V}(\mathring{x}_i) \mapsto \mathcal{V}(\underline{s}_i),\ldots\}^{\mathbf{P}}(\mathcal{V}(\mathring{p}))$$

As a special exception, $\mathcal{V}^{\mathbf{P}}\,\underline{P}$ is considered *undefined* if $\mathcal{V}(\mathring{x}_1) = \mathcal{V}(\mathring{x}_2)$ for $\mathring{x}_1 \neq \mathring{x}_2$ such that $\mathring{x}_1$ occurs in $\underline{P}$ below a form template containing a binding element $\mathtt{(}\ldots,\mathring{x}_2,\ldots\mathtt{)}$.
For example, $\{\mathtt{a} \mapsto \mathtt{x}, \mathtt{b} \mapsto \mathtt{x}, \mathtt{c} \mapsto \mathtt{x}\}$ cannot be applied to `(a).c.0 | (b).c.0`, which would otherwise capture names and produce `(x).x.0 | (x).x.0`.

$$\frac{\mathbf{reduce}\{\underline{P}_1 \hookrightarrow \underline{P}_2\} \in \mathcal{R}}{\mathcal{R} \vdash \mathcal{V}^{\mathbf{P}}\,\underline{P}_1 \hookrightarrow \mathcal{V}^{\mathbf{P}}\,\underline{P}_2} \qquad \frac{\mathcal{R} \vdash P \hookrightarrow Q}{\mathcal{R} \vdash \nu(x).P \hookrightarrow \nu(x).Q}$$

$$\frac{\mathbf{active}\{\mathring{p}\ \mathbf{in}\ \underline{P}\} \in \mathcal{R} \qquad \mathcal{R} \vdash P \hookrightarrow Q}{\mathcal{R} \vdash (\mathcal{V}[\mathring{p} \mapsto P])^{\mathbf{P}}\,\underline{P} \hookrightarrow (\mathcal{V}[\mathring{p} \mapsto Q])^{\mathbf{P}}\,\underline{P}} \qquad \frac{\mathcal{R} \vdash P \hookrightarrow Q}{\mathcal{R} \vdash P \mid R \hookrightarrow Q \mid R} \qquad \frac{P \equiv Q \qquad \mathcal{R} \vdash Q \hookrightarrow R}{\mathcal{R} \vdash P \hookrightarrow R}$$

Fig. 4. Syntax and semantics of reduction rules

to the right of the arrow in **reduce** rules, but the subset we present here is sufficient to express the calculi listed in Sect. 2.5.

As an example, with this syntax we can describe Mobile Ambients by the ruleset

$$\begin{aligned}
\mathcal{R}_{\text{MA}} = \{\ & \mathbf{active}\{\mathtt{P}\ \mathbf{in}\ \mathtt{a[P]}\}, \\
& \mathbf{reduce}\{\mathtt{a[in\,b.P \mid Q]} \mid \mathtt{b[S]} \hookrightarrow \mathtt{b[a[P \mid Q] \mid S]}\}, \\
& \mathbf{reduce}\{\mathtt{a[b[out\,a.}P \mid \mathtt{Q]} \mid \mathtt{S]} \hookrightarrow \mathtt{a[S] \mid b[P \mid Q]}\}, \\
& \mathbf{reduce}\{\mathtt{open\,a.P \mid a[R]} \hookrightarrow \mathtt{P \mid R}\}, \\
& \mathbf{reduce}\{\mathtt{<M>.P \mid (a).Q} \hookrightarrow \mathtt{P} \mid \{\mathtt{a} := \mathtt{M}\}\mathtt{Q}\}\ \}
\end{aligned}$$

These five rules are all that is necessary to instantiate Meta✶ to be Mobile Ambients.[5] The four **reduce** rules directly correspond to the reduction axioms of the target calculus. The rule **active**{P **in** a[P]} is the Meta✶ notation for the "evaluation context" rule $P \hookrightarrow P' \Longrightarrow a[P] \hookrightarrow a[P']$. This is, in fact, the only concrete **active** rule that we have so

[5] The rules are not sufficient to get communication reduction with arbitrary arity. Our implementation provides a syntax for defining arbitrary-arity communication rules, but for reasons of space and clarity we omit it in our formal development.

far needed for encoding existing calculi. We might just have hard-coded something like this rule into Meta✶, but we find it cleaner not to have any built-in distinction between "action" forms and "process container" forms in the theory.

The lower half of Figure 4 defines how to derive a reduction relation between process terms from a ruleset. For example, let $\mathcal{R}_{\text{eat}}$ be the ruleset for the fictional calculus from Fig. 1: $\mathcal{R}_{\text{eat}} = \{\ \textbf{active}\{\mathtt{P}\ \textbf{in}\ \mathtt{a}[\mathtt{P}]\}, \textbf{reduce}\{\mathtt{a}[\mathtt{eat\,b} \mid \mathtt{P}] \mid \mathtt{b}[\mathtt{Q}] \hookrightarrow \mathtt{a}[\mathtt{P} \mid \mathtt{b}[\mathtt{Q}]]\}\ \}$. We can then instantiate the first inference rule in the bottom third of Fig. 4 to obtain

$$\mathcal{R}_{\text{eat}} \vdash \mathtt{y1}[\mathtt{eat\ x} \mid \mathtt{z1}[\mathtt{0}]] \mid \mathtt{x}[\mathtt{eat\ z1} \mid \mathtt{eat\ z2}] \hookrightarrow \mathtt{y1}[\mathtt{x}[\mathtt{eat\ z1} \mid \mathtt{eat\ z2}] \mid \mathtt{z1}[\mathtt{0}]]$$

by choosing $\mathcal{V}$ to be $\{\mathtt{a} \mapsto \mathtt{y1}, \mathtt{b} \mapsto \mathtt{x}, \mathtt{P} \mapsto \mathtt{z1}[\mathtt{0}], \mathtt{Q} \mapsto (\mathtt{eat\ z1} \mid \mathtt{eat\ z2})\}$.

A reduction rule must not allow a well-scoped term to reduce to a non-well-scoped one. In order to guarantee this, the process templates in them must satisfy some scoping restrictions that are not apparent from the syntax. The restrictions will be satisfied by most rules that are intuitively sensible; because a precise understanding of how the restrictions work is not important for a high-level understanding of Meta✶, we refer to this paper's long version [17] for a precise definition.

2.5 Example Instantiations

We have checked (using machine-readable rulesets for our type inference implementation for Meta✶/Poly✶) that Meta✶ can handle π-calculus [18]; Mobile Ambients [8]; Safe Ambients [16] and various variants regarding where the $\overline{\mathsf{out}}$ capability must be found and which name co-capabilities must refer to (variants with anonymous co-capabilities also exist [15]); the Seal calculus [25] in the non-duplicating variation of [10]; Boxed Ambients [5], as well as its "asynchronous" and "Seal-like" variants (the latter being what later papers most often refer to as BA); Channelled Ambients [21]; NBA [6]; Push and Pull Ambient Calculus [22]; and $\mathbf{M^3}$ [12].

In many of these cases, Meta✶ supports the straightforward way to notate process terms as flat ASCII text, but in some cases the native punctuation of the target calculus must be changed superficially to conform to Meta✶ conventions about how a form looks. For example, the original send action $\overline{y}x$ from [18] is represented as "*y*<*x*>" (but "/*y x*" would also have worked), and "$\overline{\mathsf{enter}}(x,y)$" from [6] becomes "co-enter$(x)\,y$", because it binds x in its continuation but uses y to handshake with the entering ambient. The "$n[c_1,\ldots,c_k;P]$" construction in Channelled Ambients [21] can be represented as "$n[\mathsf{cs}.(c_1.0 \mid \cdots \mid c_k.0) \mid \mathsf{ps}.P]$". In our ruleset for Mobile Ambients with Objective Moves [7], the fact that reduction rules cannot inspect the structure of messages forces us to represent the original "*go M*.*m*[*P*]" as "go.*M*.*m*[*P*]".

3 Poly✶: Types for Meta✶

3.1 Shape Predicates

As described in the introduction, *shape predicates* are the central concept in Poly✶. A shape predicate denotes a set of process terms; certain shape predicates that are provably closed under reduction are *types*. The full language of shape predicates is somewhat complex, so let us introduce it piecewise. The basic idea of shape predicates can be explained simply: *A shape predicate looks like a process term. It matches any process*

term that can arise by repeatedly duplicating and/or removing sub-terms of the shape predicate. Here, "duplicating" and "removing" sub-terms means applying the rewriting rules $\pi \rightsquigarrow \pi \mid \pi$ and $\pi \rightsquigarrow 0$ to any syntactic subterm of the shape predicate, in addition to using the structural congruence relation for terms.

For example, a shape predicate written a[in b | in c] | c[0] would match the terms a[in b | in c] | c[0] (which is identical to the shape predicate) and a[in b] | a[in c] | c[0] (which arises by duplicating a[···] and then removing one of the in subterms in each of the copies). But a[in b] | c[a[0]] does not match, because duplicating subterms cannot make a[] appear below a c[]. Neither is in b | in c | c[0] allowed — when removing the a[] form, the entire subterm below it must be removed.

The type in Fig. 1 can be written in term shape as y1[eat x | z1[0] | x[z1[0] | eat z1 | eat z2]] | x[eat z1 | eat z2] | y2[x[eat z1 | eat z2 | z2[0]] | eat x | z2[0]].

In practice shape predicates cannot be exactly term-shaped, but it pays to keep this naive idea in mind as an intuition about what shape predicates are. When we introduce complications in the rest of this subsection, they should all be understood as "whatever is necessary to make the naive idea work in practice".

Replication ($!P$) is ignored when matching shape predicates. This is sensible because $!P$ behaves like an infinite number of P's running in parallel, and any *finite* number of P's in parallel match a shape predicate exactly if a single P does.

We want to represent all possible computational future of each term smashed together in a single shape predicate. This creates problems for the naive idea, because terms such as !x[eat x] can evolve to arbitrary deep nestings of x[···]. Therefore we need shape predicates to be *infinitely* deep trees. We restrict ourselves to infinite shape predicates with finite *representations* — in other words, regular trees.

There are several known ways of representing regular trees as linear syntax, but we have found it easier to work directly with *graphs*. A shape predicate now has the form $\langle G \mid X \rangle$, where G is a directed (possibly cyclic) graph where each edge is labelled with a form, and X is a designated root node in the graph. A term matches the shape predicate if its syntax tree can be "bent into shape" to match a subgraph such that each form in the term lies atop a corresponding edge in the graph (edges may be used more than once), and groups of parallel composition, !, and 0 lie within a single node in the graph.

The formal structure of Poly✶ uses graphs where node names are just opaque identifiers and the meaning is given by edge labels. When *displaying* the graphs (as in Fig. 1) we have found it useful to put each edge label inside the edge's target node. Of course this can't be done in the rare cases when two edges that share a target disagree.

Graphs alone are not enough to guarantee a finite type for every term. For example, the term <x> | !(y).<y.y> can (given the reduction rules of MA) evolve into terms with messages that contain arbitrarily long chains of x's *within* a single form. We need to abstract over messages such that an infinity of forms that look alike except having messages of different length within them can be described by the same shape graph label. This is the job of **message types** μ, which are defined in Figure 5.

The message type $\{f_1, \ldots, f_k\}*$ describes any message built from the any of forms f_i — *except* messages that are single names; such a message is matched by the message type $\{x\}$ instead. When $\{x\}$ is the *only* message type that matches x, we can see unambiguously from a message type whether $\bullet$ will result from trying to substitute a message

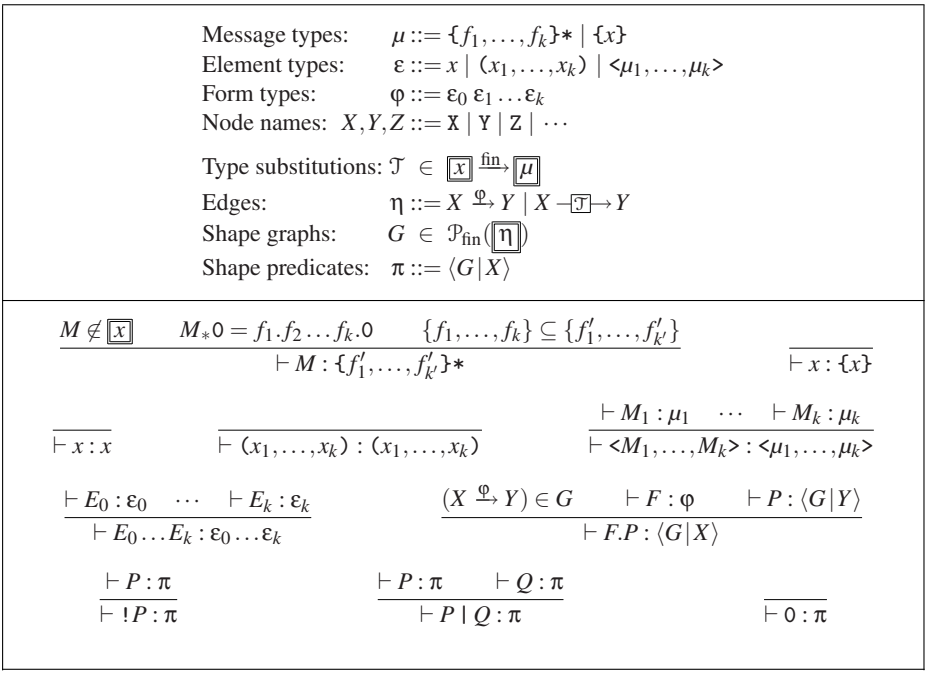


Fig. 5. The syntax and semantics of shape predicates. Edges of the form $X \xrightarrow{\boxed{\mathcal{T}}} Y$ do not influence the semantics of the shape predicate; Sect. 3.2 explains what they are for

it matches into an element position. We use **element types** ε and **form types** φ to build form-like structures out of message types and non-message elements.

The syntax and semantics of shape predicates is defined in Figure 5. To save space and present the basic theory more clearly we do not handle *name restriction*; how to treat it is described in [17]. We have also omitted a third form of message types, sequenced message types, which allow more precise types and are defined in [17–sec. 5.2].

Define the **meaning** of message/element/form types and of shape predicates by

$$[\![\mu]\!] = \{M \mid \vdash M : \mu\} \quad [\![\varepsilon]\!] = \{E \mid \vdash E : \varepsilon\} \quad [\![\varphi]\!] = \{F \mid \vdash F : \varphi\} \quad [\![\pi]\!] = \{P \mid \vdash P : \pi\}$$

Proposition 3.1. *The meanings of shape predicates respect the structural congruence: If $P \equiv Q$ then $\vdash P : \pi \iff \vdash Q : \pi$ for all π.* □

Let $\mu_1 *\mu_2$ be the least message type whose meaning includes $M.N$ for all $M \in [\![\mu_1]\!], N \in [\![\mu_2]\!]$. With the language of message types presented here (omitting sequenced message types from [17]), $\mu_1 *\mu_2$ always has the form $\{f_1,\ldots,f_k\}*$, where the f_i's are all the sub-forms that appear in either μ_1 or μ_2 in some canonical order (for this purpose the sub-form x is considered to appear in $\mu = \{x\}$). The $\mu_1 *\mu_2$ operation is associative.

3.2 Flow Edges and Subtyping

The only part of the shape predicate syntax of Figure 5 that has yet not been explained is the **flow edges** $X \xrightarrow{\boxed{\mathcal{T}}} Y$. They are not used at all in the above definition of the *meaning*

$$\mathcal{T}^{\mathbf{f}}(x_0 \dots x_k) = \begin{cases} \mathcal{T}(x_0) & \text{when } k = 0 \text{ and } x_0 \in \operatorname{Dom} \mathcal{T} \\ \{\mathcal{T}^{\mathbf{E}} x_0 \dots \mathcal{T}^{\mathbf{E}} x_k\}* & \text{otherwise} \end{cases}$$

$$\mathcal{T}^{\mathbf{M}}\{f_1, \dots, f_k\}* = \{\}*_*\mathcal{T}^{\mathbf{f}} f_{1*}\cdots_*\mathcal{T}^{\mathbf{f}} f_k \qquad \mathcal{T}^{\mathbf{M}}\{x\} = \begin{cases} \mathcal{T}(x) & \text{if } x \in \operatorname{Dom} \mathcal{T} \\ \{x\} & \text{otherwise} \end{cases}$$

$$\mathcal{T}^{\mathbf{E}} x = \begin{cases} y & \text{when } \mathcal{T}^{\mathbf{M}}\{x\} = \{y\} \text{ for some } y \\ \bullet & \text{otherwise} \end{cases}$$

$$\mathcal{T}^{\mathbf{E}} \langle \mu_1, \dots, \mu_k \rangle = \langle \mathcal{T}^{\mathbf{M}} \mu_1, \dots, \mathcal{T}^{\mathbf{M}} \mu_k \rangle$$

$$\mathcal{T}^{\mathbf{E}} (x_1, \dots, x_k) = (x_1, \dots, x_k)$$

Fig. 6. The action of type substitutions. $\mathcal{T}^{\mathbf{f}} f$ is the action on subforms, $\mathcal{T}^{\mathbf{M}} \mu$ is the action on message types, and $\mathcal{T}^{\mathbf{E}} \varepsilon$ is the action on element types

of the shape graph, but they will be important for distinguishing between types and non-types. In brief, the flow edge $X \xrightarrow{\boxed{\mathcal{T}}} Y$ asserts that there may be a reduction where a process described by X is moved to Y and in the process incurs a substitution described by $\mathcal{T}$.

Alternatively, $X \xrightarrow{\boxed{\mathcal{T}}} Y$ can be viewed as a *demand* that whenever $P \in [\![\langle G \,|\, X \rangle]\!]$ and Q arises by applying a substitution described by $\mathcal{T}$ to P, it must hold that $Q \in [\![\langle G \,|\, Y \rangle]\!]$. Because flow edges do not contribute to the meaning of shape predicates, there is no guarantee that this demand is actually satisfied for a shape predicate that contains the flow edge. This is a global property of the shape graph, and we will shortly define a class of **flow closed** shape graphs where the interpretation of flow edges is always true.

An important special case is when $\mathcal{T} = \varnothing$, where the process moves *without* any substitution. Then $X \xrightarrow{\boxed{\varnothing}} Y$ can also be viewed as an assertion that $\langle G \,|\, X \rangle$ is a *subtype* of $\langle G \,|\, Y \rangle$, or, symbolically, that $[\![\langle G \,|\, X \rangle]\!] \subseteq [\![\langle G \,|\, Y \rangle]\!]$. We therefore also speak of $X \xrightarrow{\boxed{\varnothing}} Y$ as a **subtyping edge**.

Write $\vdash \mathcal{S} : \mathcal{T}$ iff $\operatorname{Dom} \mathcal{S} = \operatorname{Dom} \mathcal{T}$ and $\vdash \mathcal{S}(x) : \mathcal{T}(x)$ for all $x \in \operatorname{Dom} \mathcal{S}$.

Define the action of type substitution on subforms and message/element types by the rules in Figure 6. This definition ensures that $[\![\mathcal{T}^{\mathbf{M}} \mu]\!]$ contains the result of every *term* substitution $\mathcal{S}^{\mathbf{M}} M$ where $\vdash \mathcal{S} : \mathcal{T}$ and $\vdash M : \mu$, and likewise for elements.

Definition 3.2. *The shape graph G is **flow closed** iff whenever G contains $X \xrightarrow{\varphi} Y$ and $X \xrightarrow{\boxed{\mathcal{T}}} Z$ such that $\mathrm{BN}(\varphi) \cap \operatorname{Dom} \mathcal{T} = \varnothing$, then there is a W such that G contains $Y \xrightarrow{\boxed{\mathcal{T}}} W$ and additionally it holds that*

1. *If $\varphi = x$ and $\mathcal{T}(x) = \{y\}$, then G contains $Z \xrightarrow{y} W$.*
2. *If $\varphi = x$ and $\mathcal{T}(x) = \{f_1, \dots, f_k\}*$, then $W = Z$ and G contains $Z \xrightarrow{f_i} Z$ for $1 \le i \le k$.*
3. *If none of the above apply and $\varphi = \varepsilon_0 \dots \varepsilon_k$, then G contains $Z \xrightarrow{\mathcal{T}^{\mathbf{E}}\varepsilon_0 \dots \mathcal{T}^{\mathbf{E}}\varepsilon_k} W$.*

We call a shape predicate $\langle G \,|\, X \rangle$ flow closed iff its G component is. □

Intuitively, a flow-closed graph is one where the intuitive meanings of flow edges are true. That is the content of the following theorem:

Proposition 3.3. *Let G be flow closed and contain $X \xrightarrow{\boxed{\mathcal{T}}} Y$. Assume that $\vdash \mathcal{S} : \mathcal{T}$ and that $\mathrm{BN}(P) \cap \operatorname{Dom} \mathcal{T} = \varnothing$. Then $\vdash P : \langle G \,|\, X \rangle$ implies $\vdash \mathcal{S}^{\mathbf{P}} P : \langle G \,|\, Y \rangle$* □

Let a **type instantiation** $\mathcal{U}$ map $\boxed{\mathring{x}}$ to $\boxed{x} \setminus \{\bullet\}$, $\boxed{\mathring{m}}$ to $\boxed{\mu}$, and $\boxed{\mathring{p}}$ to $\boxed{X}$. It applies to element and form templates completely componentwise (giving element and form types), just like term instantiations do. The relation betwen type instantiations and process templates are given by this inference system:

$$\frac{X = \mathcal{U}(\mathring{p})}{\mathcal{U} \models_{\mathsf{L}} \mathring{p} : \langle G \,|\, X\rangle} \qquad \frac{(\mathcal{U}(\mathring{p}) \xrightarrow{\boxed{\varnothing}} X) \in G}{\mathcal{U} \models_{\mathsf{R}} \mathring{p} : \langle G \,|\, X\rangle} \qquad \frac{(\mathcal{U}(\mathring{p}) \xrightarrow{\boxed{\ldots,\mathcal{U}(\mathring{x}_i)\mapsto \mathcal{U}^{\mathbf{S}}\underline{s}_i,\ldots}} X) \in G}{\mathcal{U} \models_{\mathsf{R}} \{\ldots,\mathring{x}_i := \underline{s}_i,\ldots\}\, \mathring{p} : \langle G \,|\, X\rangle}$$

$$\frac{}{\mathcal{U} \models_s \mathtt{0} : \pi} \qquad \frac{\mathcal{U} \models_s \underline{P}_1 : \pi \qquad \mathcal{U} \models_s \underline{P}_2 : \pi}{\mathcal{U} \models_s \underline{P}_1 \mid \underline{P}_2 : \pi} \qquad \frac{(X \xrightarrow{\mathcal{U}^{\mathbf{F}}\underline{F}} Y) \in G \qquad \mathcal{U} \models_s \underline{P} : \langle G \,|\, Y\rangle}{\mathcal{U} \models_s \underline{F}.\underline{P} : \langle G \,|\, X\rangle}$$

where $\mathcal{U}^{\mathbf{S}}\mathring{x} = \{\mathcal{U}(\mathring{x})\}$ and $\mathcal{U}^{\mathbf{S}}\mathring{m} = \mathcal{U}(\mathring{m})$. The rules for template processes have an L variant and an R variant; the variable letter s ranges over L and R.

As a special exception, $\mathcal{U} \models_s \underline{P} : \pi$ is *not* considered to hold if $\mathcal{U}(\mathring{x}_1) = \mathcal{U}(\mathring{x}_2)$ for $\mathring{x}_1 \neq \mathring{x}_2$ such that $\mathring{x}_1$ occurs in $\underline{P}$ below a form template containing a binding element $(\ldots,\mathring{x}_2,\ldots)$.

Fig. 7. Matching of reduction rules to shape graphs

The assumption that $\mathrm{BN}(P) \cap \mathrm{Dom}\,\mathcal{T} = \varnothing$ will be true in our uses because we are assuming that all terms are well-scoped.

3.3 Semantic and Syntactic Closure; Types

Call the shape predicate π **semantically closed** with respect to $\mathcal{R}$ iff $\vdash P : \pi$ and $\mathcal{R} \vdash P \hookrightarrow Q$ imply $\vdash Q : \pi$.

As described above, we want *types* to be semantically closed shape predicates. But it is not easy to recognise semantic closure.[6] We will therefore define a restricted, but easier to decide, class of *syntactically* closed shape predicates, which will be the types.

Figure 7 defines a way to match process templates directly to type graphs without going via the process semantics from Fig. 4. A type instantiation applies to message, element, and form templates much like the $\mathcal{V}$'s in Fig. 4, but the process part is different because a type instantiation maps process metavariables to nodes in the shape graph rather than processes. This is best illustrated with an example. Assume we wish to find out whether $\langle G_0 \,|\, \mathtt{R}\rangle$ is closed with respect to $\mathcal{R}_{\mathrm{eat}}$, where

$$G_0 = \left\{ \begin{array}{l} \mathtt{R} \xrightarrow{\mathtt{t[]}} \mathtt{X} \xrightarrow{\mathtt{p[]}} \mathtt{Y} \xrightarrow{\mathtt{eat\ q}} \mathtt{Z} \xrightarrow{\mathtt{foo}} \mathtt{T} \\ \qquad\qquad\quad \mathtt{X} \xrightarrow{\mathtt{q[]}} \mathtt{W} \qquad \mathtt{Y} \xrightarrow{\mathtt{q[]}} \mathtt{V} \xrightarrow{\mathtt{bar}} \mathtt{U} \\ \qquad\qquad\qquad\qquad\quad \mathtt{W} \xrightarrow{\mathtt{s[]}} \mathtt{S} \end{array} \right\}$$

$\mathcal{R}_{\mathrm{eat}}$ has only one **reduce** rule, and we look for matches of its left-hand side, that is $(X_0, \mathcal{U}_0)$ such that X_0 can be reached from the root by edges with labels of the shape $x\mathtt{[]}$ and $\mathcal{U}_0 \models_{\mathsf{L}} \mathtt{a[eat\,b \mid P] \mid b[Q]} : \langle G_0 \,|\, X_0\rangle$. The only such pair is

[6] E.g., let $G = \{\mathtt{Y1} \xleftarrow{\mathtt{c}} \mathtt{Y0} \underset{\mathtt{b}}{\overset{\mathtt{b}}{\rightleftarrows}} \mathtt{X0} \xrightarrow{\mathtt{(a)<\{b\}*>}} \mathtt{X1} \xrightarrow{\mathtt{a}} \mathtt{X2} \xrightarrow{\mathtt{b}} \mathtt{X3} \xrightarrow{\mathtt{a}} \mathtt{X4} \xrightarrow{\mathtt{c}} \mathtt{X5}\}$. Then $\langle G \,|\, \mathtt{X0}\rangle$ happens to be semantically closed with respect to $\{\mathbf{reduce}\{\mathtt{(a)<M>.P} \hookrightarrow \{\mathtt{a} := \mathtt{M}\}\mathtt{P}\}\}$, but it is not trivial to see this in a systematic way.

$$X_0 = \mathtt{X}, \quad \mathcal{U}_0 = \{\mathtt{a} \mapsto \mathtt{p}, \mathtt{b} \mapsto \mathtt{q}, \mathtt{P} \mapsto \mathtt{Y}, \mathtt{Q} \mapsto \mathtt{W}\}$$

For the graph to be syntactically closed, the same X and $\mathcal{U}_0$ must match the *right*-hand side of the rule, i.e. $\mathcal{U}_0 \models_{\mathtt{R}} \mathtt{a}[\mathtt{P} \mid \mathtt{b}[\mathtt{Q}]] : \langle G_0 \,|\, \mathtt{X}\rangle$. This turns out not to be true; if one tries to construct at derivation it turns out that the judgement $\mathcal{U}_0 \models_{\mathtt{R}} \mathtt{Q} : \langle G_0 \,|\, \mathtt{V}\rangle$ is needed, and that can only be true if $(\mathcal{U}_0(\mathtt{Q}) = \mathtt{W} \multimap \mathtt{V}) \in G_0$ which is not the case.

Thus the graph is not syntactically closed for $\mathcal{R}_{\text{eat}}$, and indeed it is not semantically closed either, because $P_1 = \mathtt{t}[\mathtt{p}[\mathtt{eat\ q}] \mid \mathtt{q}[\mathtt{s}[\mathtt{0}]]]$ is in $[\![\langle G_0 \,|\, \mathtt{R}\rangle]\!]$ but $P_2 = \mathtt{t}[\mathtt{p}[\mathtt{q}[\mathtt{s}[\mathtt{0}]]]]$ is not, even though $\mathcal{R}_{\text{eat}} \vdash P_1 \hookrightarrow P_2$. If we add the missing edge, giving $G_1 = G_0 \cup \{\mathtt{W} \multimap \mathtt{V}\}$, we do get $\mathcal{U}_0 \models_{\mathtt{R}} \mathtt{a}[\mathtt{P} \mid \mathtt{b}[\mathtt{Q}]] : \langle G_1 \,|\, \mathtt{X}\rangle$, but G_1 is not *flow closed*, and we need to add even more edges to make it so. In the end we get

$$G_2 = \left\{ \begin{array}{l} \mathtt{R} \xrightarrow{\mathtt{t}[]} \mathtt{X} \xrightarrow{\mathtt{p}[]} \mathtt{Y} \xrightarrow{\mathtt{eat\ q}} \mathtt{Z} \xrightarrow{\mathtt{foo}} \mathtt{T} \\ \quad\quad\quad\quad\quad\quad \mathtt{X} \xrightarrow{\mathtt{q}[]} \mathtt{W} \dashrightarrow \mathtt{V} \xrightarrow{\mathtt{bar}} \mathtt{U} \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathtt{Y} \xrightarrow{\mathtt{q}[]} \mathtt{V} \xrightarrow{\mathtt{s}[]} \mathtt{VV} \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathtt{W} \xrightarrow{\mathtt{s}[]} \mathtt{S} \dashrightarrow \mathtt{VV} \end{array} \right\}$$

where, as in Fig. 1, subtyping edges are shown as dashed lines. The shape predicate $\langle G_2 \,|\, \mathtt{R}\rangle$ is syntactically and semantically closed.

We are now ready do define syntactic closure formally:

Definition 3.4. *Let the shape predicate $\pi = \langle G | X\rangle$ be given. The set of* ***active*** *nodes for $\mathcal{R}$, written $\mathsf{active}_{\mathcal{R}}(\pi)$, is the least set A of nodes which contains X and such that for all $Y \in A$ and all* **active**$\{\mathring{p}$ **in** $\underline{P}\} \in \mathcal{R}$*, it holds that $\mathcal{U} \models_{\mathsf{L}} \underline{P} : \langle G | Y\rangle$ implies $\mathcal{U}(\mathring{p}) \in A$.* □

Definition 3.5. *G is* ***locally closed*** *at X with respect to $\mathcal{R}$ iff whenever $\mathcal{R}$ contains* **reduce**$\{\underline{P}_1 \hookrightarrow \underline{P}_2\}$ *it holds that $\mathcal{U} \models_{\mathsf{L}} \underline{P}_1 : \langle G | X\rangle$ implies $\mathcal{U} \models_{\mathsf{R}} \underline{P}_2 : \langle G | X\rangle$.* □

Definition 3.6. *The shape predicate $\pi = \langle G | X\rangle$ is* ***syntactically closed*** *with respect to $\mathcal{R}$ iff G is flow closed and also locally closed at every $X \in \mathsf{active}_{\mathcal{R}}(\pi)$. When this holds, we call π an ($\mathcal{R}$-)**type**.* □

Checking that a purported type is really syntactically closed is algorithmically easy; see [17] for details.

Theorem 3.7 (Subject reduction). *If π is syntactically closed with respect to $\mathcal{R}$, then it is also semantically closed with respect to $\mathcal{R}$.* □

3.4 What to Do with Types

Once we have a type, what can we use it for? An obvious possibility is to check whether the term may "go wrong". The user (or, more likely, the designer of a programming environment that uses Poly$\star$) specifies what "going wrong" means. It is a clear indication of error if $\bullet$ turns up in an active position, but opinions may differ about if a $\bullet$ is produced at a place in process tree that never becomes active.

One can also imagine very application-specific properties to check for, for example "this process can never evolve to a configuration where an ambient named a is inside one named b". This is easy to check for in the shape graph. Alternatively, one may want

to write this as a rule, to have Poly✶ do the checking: **reduce**$\{\mathtt{b}[\mathtt{a}[\mathtt{P}] \mid \mathtt{Q}] \hookrightarrow \bullet.0\}$. The ability to write such rules is one of the reasons why Meta✶ does not distinguish strictly between "names" and "keywords".

Poly✶ makes it fairly easy to check *safety properties* like "no unauthorised ambients (e.g., a) inside secure ambients (e.g., b)", but there are also questions of safety that Poly✶ cannot help determine. This includes properties that depend on the *order* in which things happen, such as the "correspondence assertions" often used to specify properties of communication protocols. There are type systems for process calculi that can reason about such temporal properties (for example, [13] for the π-calculus), but we are aware of none that also handle locations and ambient-style mobility.

4 Type Inference for Poly✶

Assume now that we are given a process term P and a ruleset $\mathcal{R}$; we want to produce an $\mathcal{R}$-type for P. It is trivial to construct *some* type for P – one with a single node and a lot of edges in the shape graph. However, such a type may need to contain $\bullet$'s and thus not prove that P "cannot go wrong". In this section we discuss how to automatically infer more informative types.

We do not know how to do type inference that is complete for the full Poly✶ type system; it allows too many types. Therefore we begin by defining a set of *restricted* types, for which we *can* have complete type inference.

Definition 4.1. *Write* $\varphi_1 \approx \varphi_2$ *iff* $[\![\varphi_1]\!] \cap [\![\varphi_2]\!] \neq \varnothing$. □

The $\approx$ relation is close to being equality. The only way for two non-identical φ's to be related by $\approx$ is if they contain message types of the shape $\{\cdots\}*$. It is relatively safe to imagine the $\approx$ is just a fancy way to write $=$, at least to a first approximation.

Definition 4.2. *G satisfies the* ***width restriction*** *iff whenever it contains* $X \xrightarrow{\varphi} Y$ *and* $X \xrightarrow{\varphi'} Y'$ *with* $\varphi \approx \varphi'$*, it holds that* $Y = Y'$. □

Definition 4.3. *G satisfies the* ***depth restriction*** *iff whenever it contains a chain* $X_0 \xrightarrow{\varphi_1} X_1 \xrightarrow{\varphi_2} \cdots \xrightarrow{\varphi_k} X_k$ *with* $\varphi_1 \approx \varphi_k$*, it holds that* $X_1 = X_k$. □

Our type inference algorithm only produces types that satisfy both restrictions. The width restriction means that when type inference needs to add an outgoing edge from a node in the graph, it never has to choose between reusing an existing edge starting there and creating a new edge with a fresh target node, because the latter is forbidden by the restriction when there is any reusable edge. The depth restriction bounds the length of a simple path in a shape graph that can be constructed with a given set of names and a given maximal form arity, and so also bounds the total number of nodes in the graph. Therefore the closing operation described below cannot keep adding edges and nodes to the graph indefinitely and will eventually stop. (These two restrictions replace the notions of "discrete" and "modest" types in [2], which sometimes admitted slightly more precise types, but were very complex and hard to understand).

In [17] we describe a feature of our implementation which allows it to loosen the two restrictions by tracking the origin of each φ in the type graph.

The type inference proceeds in two phases. First we construct a minimal shape predicate which the term matches. Then we **close** the shape predicate — that is, rewrite its shape graph as necessary to make it syntactically closed.

The initial phase is simple. Because shape predicates "look like terms", we can just convert the abstract syntax tree of the term to a tree-shaped shape graph. This graph may or may not satisfy the width and depth restrictions. If it does not, unify the nodes that must be equal.[7] That may cause further violations of the two restrictions; continue unifying nodes as necessary until the restrictions are satisfied.

The closing of the shape graph is where the real work of type inference happens. It happens in a series of steps. In each step, check whether the shape graph is syntactically closed. If it is, the algorithm ends. Otherwise, the lack of closure can only be because edges already in the graph imply that some other edges *ought* to exist (by Definitions 3.2 or 3.5) but do not. In that case, add the new nodes and edges required by the failing rule, and do another round of unifications to enforce the width and depth restrictions.

The width and depth restriction together guarantee that the closure phase terminates. We do not have any good worst-case complexity bounds for the closure phase; instead our implementation allows further restrictions on types to be applied in order to quench blow-ups one might observe with particular calculi and example terms. The tightest restrictions will enforce polynomial complexity, at the cost of losing the possibility of spatial polymorphism. Thus restricted, Poly$\ast$ has a strength roughly comparable to current non-polymorphic type systems for ambient-derived calculi.

Theorem 4.4 (Principal typings). *A result of type inference π is a principal typing [26] for the input term P: For every π' such that $\vdash P : \pi'$ it holds that $[\![\pi']\!] \supseteq [\![\pi]\!]$.* □

4.1 Implementation

We have implemented our type inference algorithm. Our implementation is available at the URL ⟨`http://www.macs.hw.ac.uk/DART/software/PolyStar/`⟩, as both a source download and an interactive web interface.

Beyond the features in this paper, our implementation allows fine-tuning of the analysis precision, which influences inference speed as well as inferred type size.

5 Conclusion

Poly$\ast$ extends basic properties of our previous system PolyA to a more general setting:

1. Poly$\ast$ has *subject reduction*. Also, given a process term *P* and a shape predicate π, one can decide by checking purely local conditions whether π is a *type*, and it is similarly decidable whether *P matches* π. Thus, it is decidable whether a process belongs to a specific type.

[7] This is the only way to reach a graph that satisfies the restrictions. If the width and depth restrictions had used $=$ instead of $\approx$, there might also have been the option of rewriting a φ to something larger but different, but there would not be a unique "best way" of doing that.

2. Poly✶ supports a notion of *spatial polymorphism* that achieves what Cardelli and Wegner [9] called "the purest form of polymorphism: the same object or function can be used uniformly in different type context without changes, coercions or any kind of run-time tests or special encodings of representations".
3. The types of Poly✶ are sufficiently precise that many interesting *safety/security properties* can be checked, especially those that can be formulated as questions on the possible configurations that can arise at run-time.

In addition, this paper makes these completely novel contributions:

4. Meta✶ is a syntactic framework that can be instantiated into a large family of mobile process calculi by supplying reduction rules.
5. The *generic type system* Poly✶ works for any instantiation of Meta✶. We have checked that it works for π-calculus, a large number of ambient calculi, and a version of the Seal calculus. In [2] we claimed PolyA would be easy to extend to ambient-like calculi by hand, but extending the proofs for PolyA manually would be tedious. With Meta✶ we have developed the theory to do such extensions fully automatically.
6. For the subsystem of Poly✶ satisfying the *width* and *depth* restrictions, there is a type inference algorithm (which we have implemented) that always successfully infers a *principal type* for any process term. This means that Poly✶ has the potential for *compositional analysis*.
7. The width and depth restriction are more natural and intuitive than the "discreteness" and "modesty" properties with respect to which we showed existence of principal types for PolyA.
8. Poly✶'s handling of *communication* and *substitution* has been redesigned to be more direct and intuitive than in PolyA.

5.1 Related Work

Another generic type system for process calculi was constructed by Igarashi and Kobayashi [14]. Like the shape predicates in Poly✶, their types look like process terms and stand for sets of structurally similar processes. Beyond that, however, their focus is different from ours. Their system is specific to the π-calculus and does not handle locations or ambient-style mobility. On the other hand, it is considerably more flexible than Poly✶ within its domain and can be instantiated to do such things as deadlock and race detection which are beyond the capabilities of Poly✶.

Yoshida [27] used graph types much like our shape predicates to reason about the order of messages exchanged on each channel in the π-calculus. Since this type system reasoned about *time* rather than *location*, it is not directly comparable to Poly✶, despite the rather similar type structure.

The spatial analysis of Nielson et al. [19] produces results that somewhat resemble our shape graphs, but does not have spatial polymorphism.

References

[1] M. Abadi, A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inform. & Comput.*, 148(1), 1999.

[2] T. Amtoft, H. Makholm, J. B. Wells. PolyA: True type polymorphism for Mobile Ambients. In *IFIP TC1 3rd Int'l Conf. Theoret. Comput. Sci. (TCS '04)*. Kluwer Academic Publishers, 2004.

[3] T. Amtoft, F. Turbak. Faithful translations between polyvariant flows and polymorphic types. In *Programming Languages & Systems, 9th European Symp. Programming*, vol. 1782 of *LNCS*. Springer-Verlag, 2000.

[4] G. Boudol. The π-calculus in direct style. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, 1997.

[5] M. Bugliesi, G. Castagna, S. Crafa. Boxed ambients. In *4th International Conference on Theoretical Aspects of Computer Science (TACS'01)*, vol. 2215 of *LNCS*. Springer-Verlag, 2001.

[6] M. Bugliesi, S. Crafa, M. Merro, V. Sassone. Communication interference in mobile boxed ambients. In *FST & TCS 2002*, 2002.

[7] L. Cardelli, G. Ghelli, A. D. Gordon. Mobility types for mobile ambients. In J. Wiedermann, P. van Emde Boas, M. Nielsen, eds., *ICALP'99*, vol. 1644 of *LNCS*. Springer-Verlag, 1999. Extended version appears as Microsoft Research Technical Report MSR-TR-99-32, 1999.

[8] L. Cardelli, A. D. Gordon. Mobile ambients. In M. Nivat, ed., *FoSSaCS'98*, vol. 1378 of *LNCS*. Springer-Verlag, 1998.

[9] L. Cardelli, P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), 1985.

[10] G. Castagna, G. Ghelli, F. Z. Nardelli. Typing mobility in the Seal calculus. In K. G. Larsen, M. Nielsen, eds., *CONCUR*, vol. 2154 of *LNCS*. Springer-Verlag, 2001.

[11] S. Chaki, S. K. Rajamani, J. Rehof. Types as models: Model checking message-passing programs. In *Conf. Rec. POPL '02: 29th ACM Symp. Princ. of Prog. Langs.*, 2002.

[12] M. Coppo, M. Dezani-Ciancaglini, E. Giovannetti, I. Salvo. M3: Mobility types for mobile processes in mobile ambients. In *CATS 2003*, vol. 78 of *ENTCS*, 2003.

[13] A. D. Gordon, A. S. A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoret. Comput. Sci.*, 300(1–3), 2003.

[14] A. Igarashi, N. Kobayashi. A generic type system for the pi-calculus. In *Conf. Rec. POPL '01: 28th ACM Symp. Princ. of Prog. Langs.*, 2001.

[15] F. Levi, C. Bodei. A control flow analysis for safe and boxed ambients. In *Programming Languages & Systems, 13th European Symp. Programming*, vol. 2986 of *LNCS*. Springer-Verlag, 2004.

[16] F. Levi, D. Sangiorgi. Controlling interference in ambients. In *POPL'00, Boston, Massachusetts*. ACM Press, 2000.

[17] H. Makholm, J. B. Wells. Instant polymorphic type systems for mobile process calculi: Just add reduction rules and close. Technical Report HW-MACS-TR-0022, Heriot-Watt Univ., School of Math. & Comput. Sci., 2004.

[18] R. Milner, J. Parrow, D. Walker. A calculus of mobile processes. *Inform. & Comput.*, 100(1), 1992.

[19] H. R. Nielson, F. Nielson, H. Pilegaard. Spatial analysis of BioAmbients. In R. Giacobazzi, ed., *Static Analysis: 11th Int'l Symp.*, vol. 3148 of *LNCS*, Verona, Italy, 2004. Springer-Verlag.

[20] J. Palsberg, C. Pavlopoulou. From polyvariant flow information to intersection and union types. *J. Funct. Programming*, 11(3), 2001.

[21] S. M. Pericas-Geertsen. *XML-Fluent Mobile Ambients*. PhD thesis, Boston University, 2001.

[22] I. Phillips, M. G. Vigliotti. On reduction semantics for the push and pull ambient calculus. In *Theoretical Computer Science: 2nd IFIP Int'l Conf.*, vol. 223 of *IFIP Conference Proceedings*. Kluwer, 2002.

[23] G. Sander. Graph layout through the VCG tool. In R. Tamassia, I. G. Tollis, eds., *Graph Drawing: DIMACS International Workshop, GD '94*, vol. 894 of *LNCS*. Springer-Verlag, 1994.

[24] D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995. Report no ECS-LFCS-96-345.

[25] J. Vitek, G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, vol. 1686 of *LNCS*. Springer-Verlag, 1999.

[26] J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, vol. 2380 of *LNCS*. Springer-Verlag, 2002.

[27] N. Yoshida. Graph types for monadic mobile processes. In *Foundations of Software Technology and Theoret. Comput. Sci., 16th Conf.*, vol. 1180 of *LNCS*. Springer-Verlag, 1996.

Towards a Type System for Analyzing JavaScript Programs

Peter Thiemann

Universität Freiburg
http://www.informatik.uni-freiburg.de/~thiemann

Abstract. JavaScript is a popular language for client-side web scripting. It has a dubious reputation among programmers for two reasons. First, many JavaScript programs are written against a rapidly evolving API whose implementations are sometimes contradictory and idiosyncratic. Second, the language is only weakly typed and comes virtually without development tools.

The present work is a first attempt to address the second point. It does so by defining a type system that tracks the possible traits of an object and flags suspicious type conversions. Because JavaScript is a classless, object-based language with first-class functions, the type system must include singleton types, subtyping, and first class record labels. The type system covers a representative subset of the language and there is a type soundness proof with respect to an operational semantics.

Keywords: dynamic type systems, program analysis, objects, functions.

1 Introduction

JavaScript has originally been developed by Brendan Eich at Netscape Corp as a language for client-side web scripting. Judging from the material available, the language has grown evolutionary by demand of its users. As the language gained widespread use and with the involvement of several industrial players, a language definition was published as a standard under the name ECMAScript[4].

JavaScript is a weakly typed object-based language in the sense that it has objects but no classes. It relies on prototyping instead of inheritance to share and extend functionality in the tradition of the Self language [14]. In addition to the usual support for imperative programming, JavaScript has lexically scoped first-class functions so it may be called a functional programming language, too.

To date, JavaScript has numerous users because it is the primary scripting language for dynamic HTML and it is supported by virtually all web browsers. However, programmers do not really appreciate the language. Their main problem is that programs that work with one brand of browser do not work with another brand. In fact, numerous (cook-) books and lots of code have been written to detect and address these problems. A closer look reveals that this point is not criticizing the language but rather the differences in the object hierarchies

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 408–422, 2005.

provided by different browsers. The language itself is stable since 1999 so that only obsolete browsers implement the language in a significantly different way.

Another point deplored by JavaScript programmers is the lack of development tools. Despite the fact that significant libraries and applications have been developed, it has taken until very recently that a debugger is available. Taken together with the differences in the object hierarchies, it can become very hard to debug and maintain JavaScript programs.

To a large extent, the maintenance problem is caused by the weak type system of the language. Although every value of the language has a fixed type at runtime, values are converted automatically from one type to another when the context of use requires it. While many scripting programmers advocate such weak type systems because of the convenience they offer, it is easy to find situations where the automatic conversion leads to surprising results (see Section 2).

We believe that a type-based program analyzer for JavaScript programs would be of interest to developers and maintainers alike. It would make maintenance easier by automatically inferring a type signature as a minimal interface for each function. It would make development easier by rejecting programs with type mismatches outright and by flagging suspicious conversions. It would also make multi-browser development easier because the differences between browsers can be modeled by providing different types for the built-in object hierarchies. Last but not least, it would provide a basis for higher program structuring methods like module systems.

The present work presents a first step towards the construction of such a type system. After discussing some motivating examples in Section 2, we define the syntax of a small, but paradigmatic subset of JavaScript in Section 3. Next, in Section 4, we define the syntax of a suitable type language and typing rules along with an informal description of the respective language construct. Section 5 specifies a small-step operational semantics for the language (extracted from the 160+ page verbal specification) and Section 6 presents a type soundness result. Finally, we discuss related work (Section 7) and conclude.

2 Motivation

The object is the main data structure in JavaScript. Unlike structs in C or objects in Java, a JavaScript object is not statically fixed to a certain number of properties (fields) or even to certain names of properties. An object is rather a finite function from strings to JavaScript values that can be modified in any conceivable way: properties may be dynamically added, removed, or changed. Working with objects has a few pitfalls illustrated with the following transcript (running the JavaScript interpreter Rhino [11]).

```
js> var obj = { x: 1}
js> obj.x
1
js> obj.y
js> print(obj.y)
undefined
```

The first line creates an object with property `x` and value `1`. This property can be accessed with the usual dot notation. Next, we try to access a non-existent property. Instead of failing, the result is the value `undefined` which can even be converted to a string if required by the context (last line). **Our type system can identify the places where such conversions happen.**

Since objects are really functions from strings to values, there is an alternative notation for object access which looks like array access. That is, if there is an object access in a program, the actual property name may be statically unknown.

```
js> var x = "x"
js> obj[x]
1
js> obj["undefined"] = "gotcha"
js> obj[obj.y]
```

(We leave it to the reader to figure out the answer of the interpreter to the last input.) **Our type system uses singleton types to track the values of base type variables to increase the precision of property accesses and updates.**

To evaluate the expression `a.x = 51` in a sensible way, the value of `a` must be an object. This requirement is enforced by JavaScript, but in a potentially surprising way:

```
js> var a = "black hole"
js> a.x = 51
51
js> a.x
js>
```

What is going on? The value in variable `a` has type string and string happens to be a base type. Since base type values do not possess properties, the assignment to property `x` might just fail. Instead, JavaScript creates a wrapper object for the string and creates a property `x` for it. Since `a` refers directly to the string, the wrapper object becomes garbage and the evaluation of `a.x` creates a new wrapper object which does not know anything about property `x`. **Our type system flags such silent wrapper conversions.**

If we were to start the above script with `var a = new String("black hole")` everything would work as expected and `a` could be used in the same way as a base type string value.

```
js> var a = new String("black hole")
js> a.x = 51
51
js> a.x
51
```

Auxiliary		
str	$\in$ *String Constants*	
Expressions		
e	::= `this`	self reference in method calls
	$\mid x$	variable
	$\mid c$	constant (number, string, boolean)
	$\mid$ `{`$str : e, \ldots$`}`	object literal
	$\mid$ `function` $x(x, \ldots)$`{var` $x, \ldots; s$`}`	function expression
	$\mid e[e]$	property reference
	$\mid$ `new` $e(e, \ldots)$	object creation
	$\mid e(e, \ldots)$	function call
	$\mid e$ `=` e	assignment
	$\mid p(e, \ldots)$	primitive operators (addition, etc.)
Statements		
s	::= `skip`	no operation
	$\mid e$	expression statement
	$\mid s$`;`s	sequence
	$\mid$ `if (`e`) then {`s`} else {`s`}`	conditional
	$\mid$ `while (`e`) {`s`}`	iteration
	$\mid$ `return` e	function return

Fig. 1. Syntax of Core JavaScript

3 Core JavaScript

The starting point of our work is a restricted version of the JavaScript language. It is still sufficiently rich to expose the important points in the design of a meaningful type system for the language.

JavaScript is a weakly and dynamically typed, object-based language. The layout of an object is not fixed, rather an object is a dynamic table that maps property names (strings) to property values. Although each value possesses a type, there are exhaustive automatic conversions defined between each pair of types. As there are no classes, there is instead a prototyping mechanism that allows to inherit properties from prototype objects. The language includes first-class functions which also serve as pre-methods. When a function is assigned to a property of an object, the function becomes a method and each reference to `this` in its body resolves to a reference to the object at method invocation time.

Figure 1 summarizes the syntax of Core JavaScript. There are expressions e and statements s. Expressions comprise the following alternatives

- `this` is the reference to the object receiving a method call. It only makes sense in functions used as methods or constructors.
- x variables in the usual way.
- c literals. Each literal has a type given by a function *TypeOf*.
- An object literal creates a new object with properties given by literal strings *str* as property names and their values given by expressions.[1]

[1] JavaScript also allows number literals and identifiers as property names.

- A function expression `function` $f(x_1,\ldots)$`{var` $y_1,\ldots;s$`}` defines an anonymous function with arguments $x_1,\ldots$, local variables $y_1,\ldots$, and body s. The resulting function may be recursive if the function body refers to the functions identifier f, which is *not* visible outside the function body.
- $e_1[e_2]$ is a property reference. e_1 should evaluate to an object and e_2 to a value for naming the property. The name of the property is found by converting the value of e_2 to a string. The full language allows the expression $e\,.\,x$ where the identifier x is the property name. However, this expression is equivalent to $e[\text{“}x'']$.
- `new` $e_0(e_1,\ldots)$ creates a new object and applies the function value of e_0 with parameters $e_1,\ldots$ as a method to the object. In general, it returns the newly constructed object.
- $e_0(e_1,\ldots)$ applies the function value of e_0 to the values of $e_1,\ldots$. If e_0 is a property reference, *i.e.*, $e_0 = e_{01}[e_{02}]$, then the function is called as a method.
- e_0 `=` e_1 evaluates e_0 as a reference and assigns the value of e_1 to it.
- $p(e_1,\ldots)$ stands for a primitive function call, *e.g.*, an arithmetic operation, comparison, and so on.

Functions are used in two non-standard ways in JavaScript. First, they play the role of pre-methods. A function that is assigned to a property becomes a method. Calling a function via a property reference amounts to a method invocation that additionally binds `this` to the receiver object. Second, they become constructor functions when invoked through the `new` operator. In a constructor function, `this` is bound to the new object and `new` returns whatever the function returns (or `this` if it returns `undefined`).

The full language has further kinds of expressions. There are array literals which behave similarly to object literals, assigning primitives like pre- and post-increment, property deletion, and conditional expressions. We have also “cleaned up” the syntax of function bodies: all local variable declarations are gathered at the beginning, whereas the full language allows `var` declarations everywhere in the body.

The language of statements comprises the traditional constructs. The full language includes three further variations of loops, labeled `continue` and `break` statements, a `switch` statement and a `with` statement for opening an object as the innermost scope. They are left out because they are easy to simulate with the remaining constructs. Exceptions in the style of Java are present in the full language, but are left out of our consideration because their type-based analysis adds considerable complication but not much novelty [12, 16]. Finally, JavaScript has a `var` statement to declare a variable as local in the current function. This statement may occur anywhere in the body of a function and affects all occurrences of the declared variable even in front of the declaration. Our syntax assumes that a semantic analysis already collected the local variables at the beginning of each function body.

Types
$\tau ::= \sum_{i \in T, T \subseteq \{\bot,u,b,s,n,o\}} \varphi_i$
Rows
$\varrho ::= str : \tau, \varrho$
$\quad | \quad \delta\tau$
Type environments
$\Gamma ::= \Gamma(x : \tau)$
$\quad | \quad \emptyset$

Type summands and indices
$\varphi_\bot ::= \texttt{Undef}$
$\varphi_u ::= \texttt{Null}$
$\varphi_b ::= \texttt{Bool}(\xi_b)$
$\xi_b ::= \texttt{false} \mid \texttt{true} \mid \top$
$\varphi_s ::= \texttt{String}(\xi_s)$
$\xi_s ::= str \mid \top$
$\varphi_n ::= \texttt{Number}(\xi_n)$
$\xi_n ::= num \mid \top$
$\varphi_f ::= \texttt{Function}\ (\texttt{this} : \tau; \varrho \to \tau)$
$\varphi_o ::= \texttt{Obj}(\sum_{i \in T, T \subseteq \{b,s,n,f,\bot\}} \varphi_i)(\varrho)$

Fig. 2. Syntax of types

4 Types for Core JavaScript

What is the purpose of a type system for a language like JavaScript? The usual goal of a type system is to avoid runtime errors due to type mismatches. However, such runtime errors are fairly rare in JavaScript since most of the time there is a suitable conversion function to reconcile a type mismatch. But three causes for runtime errors remain and they must be rejected by our type system.

1. Function calls and the `new` expression both expect their first operand to evaluate to a function. If that is not the case, they raise an exception.
2. Applying an arithmetic operator to an object raises an exception unless the object is a wrapper object for a number.
3. Accessing a property of the `null` object (available as a literal constant) or of the value `undefined` raises an exception.
4. Accessing a non-existent variable causes an exception.

Section 2 has also demonstrated type conversions which are convenient in some circumstances but may cause unexpected results.

4.1 Syntax of Types

Figure 2 defines the type language. The language definition[4] prescribes a value structure consisting of the base types undefined, null, boolean, number, and string as well as different kinds of objects (plain objects, wrapper objects for boolean, number, and string, function objects, array objects, and regular expression objects). Hence, the type system is based on discriminative sum types which are used at two levels. The outer level distinguishes between the different base types and objects, whereas the inner level distinguishes different features of objects. The feature component indicates the type of the implicit *VALUE* property of an object. Besides this feature component, the object type reflects the properties of an object by a row type as its second component. Row types ϱ are slightly unusual because their labels are string constants. However, this

choice is dictated by the language definition which states that property names are strings and that every value that is used for addressing a property is first converted to a string. The $\delta\tau$ at the end of a row provides a default type for all properties not mentioned explicitly. Rows are considered modulo the equations

$$str_1 : \tau_1, \ldots, str_n : \tau_n, \delta\tau = str_1 : \tau_1, \ldots, str_n : \tau_n, str : \tau, \delta\tau$$
$$\ldots, str_i : \tau_i, \ldots, str_j : \tau_j, \ldots = \ldots, str_j : \tau_j, \ldots, str_i : \tau_i, \ldots$$

where $n \in \mathbf{N}$, $str \notin \{str_1, \ldots, str_n\}$, and all str_i are different.

The type syntax clearly shows that there are wrapped and unwrapped versions of booleans, strings, and numbers. The types for `null` and objects are intrinsically wrapped (or unwrapped), and functions are always wrapped into an object. We write $\bot$ for the empty sum. Base types are further refined by *type indices*. Our definition allows as index either a constant of the appropriate type or $\top$. A constant refines the type to a singleton type whereas $\top$ does not impose a restriction. Singleton string types serve to address the properties of an object.

Each function may take `this` as an implicit parameter if the function is used as a method or a constructor. Furthermore, the number of formal parameters in a function definition need not match the number of actual parameters. If there are too few parameters, the remaining are taken to be `undefined`. If there are too many, the excess ones are ignored. Alternatively, the function body can access the actual parameter list via an array, the object `arguments`. Hence, the function type has a special place for `this` and requires an row type in place of the argument types.

4.2 Subtyping

It is rather delicate to choose a subtyping relation for Core JavaScript. Our design keeps subtyping separate from type conversion, which is modeled by a matching relation. While subtyping is vital to model the dynamically typed nature of the language, applying a conversion too early may result in rejecting a perfectly legal program.

The subtyping relation $<:$ consists of the usual subtyping for variants in the form of type summands, function arguments are dealt with in the usual contravariant manner, and rows are only subject to depth subtyping.

All elimination rules rely on the matching relation $\trianglerighteq$. This relation makes sure that a type can be converted to the form desired by the elimination. Matching is explained with the typing rule for $e_0[e_1]$ in the next subsection.

4.3 Typing Rules

The type system defines a number of judgments.

- $\Gamma \vdash e : \tau$ types an expression e in a context where its value is required.
- $\Gamma \vdash_{ref} e : \tau/\tau'$ types an expression e in a context where a reference may be required. In such a context, τ' is the type of the base object of the reference. For example, if $e = e_1[e_2]$ then e_1 is the base object. The type of the base object is required to provide the type of `this` for a method call.

- $\Gamma \vdash_{lhs} e : \tau$ types a left-hand side use in an assignment.
- $\Gamma \vdash_{stm} s \rhd \tau$ types a statement that may return a value of type τ.
- $\vdash_{acc} \varrho \mathbin{@} \tau \mapsto \tau'$ computes the type for an object access.
- $\vdash_{upd} \varrho \mathbin{@} \tau \hookleftarrow \tau'$ computes the type for an object update.

There are only two typing rules for expressions in a value context, a subtyping rule and a rule that delegates the actual work to the $\vdash_{ref}$ judgment by ignoring the base type.

$$\frac{\Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash e : \tau'} \qquad \frac{\Gamma \vdash_{ref} e : \tau/\tau'}{\Gamma \vdash e : \tau}$$

The next set of rules considers expressions (potentially) in a reference context. The rules for variables and constants are standard. Of course, neither of them provides a base object so the base type is $\bot$ in both cases.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash_{ref} x : \tau/\bot} \qquad \Gamma \vdash_{ref} c : \mathit{TypeOf}(c)/\bot$$

The typing for object literals is similar to the corresponding rule in Abadi and Cardelli's object calculus [1]. The main difference is in the choice of literal strings instead of labels. The object constructed by the object literal has no special features as indicated by the feature `Undef`.

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash_{ref} \{str_1 : e_1, \ldots, str_n : e_n\} : \mathtt{Obj}(\mathtt{Undef})(str_1 : \tau_1, \ldots, str_n : \tau_n)/\bot}$$

Function expressions are not as straightforward as usual.

$$\frac{\begin{array}{c}\Gamma(\mathtt{this} : \tau_0)(f : \tau)(x_1 : \tau_1) \ldots (x_n : \tau_n)(\mathtt{arguments} : \mathtt{Obj}(\bot)(\varrho)) \\ (y_1 : \tau_1') \ldots (y_n : \tau_n') \vdash_{stm} s \rhd \tau' \\ \tau_0 = \mathtt{Obj}(\varphi')(\varrho') \qquad \tau = \mathtt{Obj}(\mathtt{Function}\ (\mathtt{this} : \tau_0; \varrho \to \tau'))(\delta\bot) \\ \varrho = \text{"}\mathtt{length}\text{"} : \mathtt{Number}(n), \lceil 0 \rceil : \tau_1, \ldots, \lceil n-1 \rceil : \tau_n, \delta\bot\end{array}}{\Gamma \vdash_{ref} \mathtt{function}\ f(x_1, \ldots, x_n)\{\mathtt{var}\ y_1, \ldots, y_m; s\} : \tau/\bot}$$

Besides providing the type assumptions for the arguments and the function, it also establishes the type assumption for `this` and `arguments`. The latter is set up as an array, that is, an object with a `length` property and numeric properties where $\lceil n \rceil$ stands for the string containing the decimal representation of n.

The next rule concerns property access. It is the only rule that creates a meaningful reference in the judgment for reference contexts.

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 \trianglerighteq \mathtt{Obj}(\varphi_1)(\varrho_1) \quad \Gamma \vdash e_2 : \tau_2 \quad \vdash_{acc} \varrho_1 \mathbin{@} \tau_2 \mapsto \tau'}{\Gamma \vdash_{ref} e_1[e_2] : \tau'/\tau_1}$$

As e_1 denotes the base object, τ_1 is the required type of the enclosing object. The auxiliary judgment $\vdash_{acc} \varrho \mathbin{@} \tau \mapsto \tau'$ computes the type of a property stored in an object of type ϱ and accessed with a property name of type τ. Figure 3 contains its definition as well as the definition of its cousin $\vdash_{upd} \varrho \mathbin{@} \tau \hookleftarrow \tau'$ which governs the typing of update operations. Both traverse the row ϱ in one of two

$$\frac{\vdash_{acc} \varrho \mathbin{@} \tau' \mapsto \tau \quad \neg(str \in \tau') \quad \neg(str \,\mathtt{is}\, \tau')}{\vdash_{acc} str : \tau_2, \varrho \mathbin{@} \tau' \mapsto \tau} \qquad \frac{\vdash_{upd} \varrho \mathbin{@} \tau' \leftrightarrow \tau \quad \neg(str \in \tau') \quad \neg(str \,\mathtt{is}\, \tau')}{\vdash_{upd} str : \tau_2, \varrho \mathbin{@} \tau' \leftrightarrow \tau}$$

$$\frac{\vdash_{acc} \varrho \mathbin{@} \tau' \mapsto \tau_1 \quad str \in \tau' \quad \neg(str \,\mathtt{is}\, \tau') \quad \tau_1 <: \tau \quad \tau_2 <: \tau}{\vdash_{acc} str : \tau_2, \varrho \mathbin{@} \tau' \mapsto \tau} \qquad \frac{\vdash_{upd} \varrho \mathbin{@} \tau' \leftrightarrow \tau \quad str \in \tau' \quad \neg(str \,\mathtt{is}\, \tau') \quad \tau <: \tau_2}{\vdash_{upd} str : \tau_2, \varrho \mathbin{@} \tau' \leftrightarrow \tau}$$

$$\frac{str \,\mathtt{is}\, \tau'}{\vdash_{acc} str : \tau, \varrho \mathbin{@} \tau' \mapsto \tau} \qquad \frac{str \,\mathtt{is}\, \tau'}{\vdash_{upd} str : \tau, \varrho \mathbin{@} \tau' \leftrightarrow \tau}$$

$$\vdash_{acc} \delta\tau \mathbin{@} \tau' \mapsto \tau \qquad \frac{\tau <: \tau_2}{\vdash_{upd} \delta\tau_2 \mathbin{@} \tau' \leftrightarrow \tau}$$

$$str \,\mathtt{is}\, \mathtt{String}(str) \qquad str \in \mathtt{String}(\xi_s) + \ldots \quad \textit{if } str \le \xi_s$$

$$str \,\mathtt{is}\, \mathtt{Obj}(\mathtt{String}(str))(\varrho) \qquad str \in \mathtt{Obj}(\mathtt{String}(\xi_s) + \ldots)(\varrho) + \ldots \quad \textit{if } str \le \xi_s$$

$$\lceil n \rceil \,\mathtt{is}\, \mathtt{Number}(n) \qquad \lceil n \rceil \in \mathtt{Number}(\xi_n) + \ldots \quad \textit{if } n \le \xi_n$$

$$\lceil n \rceil \,\mathtt{is}\, \mathtt{Obj}(\mathtt{Number}(n))(\varrho) \qquad \lceil n \rceil \in \mathtt{Obj}(\mathtt{Number}(\xi_n) + \ldots)(\varrho) + \ldots \quad \textit{if } n \le \xi_n$$

$$\lceil \mathtt{false} \rceil \,\mathtt{is}\, \mathtt{Bool}(\mathtt{false}) \qquad \lceil \mathtt{false} \rceil \in \mathtt{Bool}(\xi_b) + \ldots \quad \textit{if } \mathtt{false} \le \xi_b$$

$$\lceil \mathtt{true} \rceil \,\mathtt{is}\, \mathtt{Bool}(\mathtt{true}) \qquad \lceil \mathtt{true} \rceil \in \mathtt{Bool}(\xi_b) + \ldots \quad \textit{if } \mathtt{true} \le \xi_b$$

$$\lceil \mathtt{false} \rceil \,\mathtt{is}\, \mathtt{Obj}(\mathtt{Bool}(\mathtt{false}))(\varrho) \qquad \text{where } x \le x \text{ and } x \le \top, \text{ for } x \in \xi.$$

$$\lceil \mathtt{true} \rceil \,\mathtt{is}\, \mathtt{Obj}(\mathtt{Bool}(\mathtt{true}))(\varrho)$$

Fig. 3. Property access

modes: If the type τ contains definite information about the property name, then τ' becomes the type of that property (or the type associated to undefined properties). If τ does not contain definite information, then the judgment returns a supertype (subtype for $\vdash_{upd}$) of all applicable property types.

The match relation $\tau \trianglerighteq \varphi$ performs a one-way matching of an arbitrary type τ to a type summand φ. Matching succeeds if τ as a whole can be converted to a type with single summand φ. It acts like a constraint in the sense that it propagates information from τ to φ but not the other way round. Its conversions correspond to the definition in Chapter 9 of the ECMAScript standard [4].

The next rule deals with function call and method invocation.

$$\frac{\tau_0 \trianglerighteq \mathtt{Obj}(\mathtt{Function}\ (\mathtt{this} : \tau'; \lceil 0 \rceil : \tau_1, \ldots, \lceil n-1 \rceil : \tau_n, \varrho \to \tau))(\varrho') \quad \Gamma \vdash_{ref} e_0 : \tau_0/\tau' \quad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash_{ref} e_0(e_1, \ldots, e_n) : \tau/\bot}$$

The rule requires that the function's argument types coincide with the types of the actual function arguments. In addition, the $\vdash_{ref}$ judgment retrieves the type τ' of a suitable base object ($\tau' = \bot$ if no such object exists). For a method invocation, τ' is the type of the receiver object, `this`.

The rule for `new` covers the use of a function as a constructor.

$$\frac{\tau_0 \trianglerighteq \mathtt{Obj}(\mathtt{Function}\ (\mathtt{this} : \tau''; \lceil 0 \rceil : \tau_1, \ldots, \lceil n-1 \rceil : \tau_n, \varrho \to \tau))(\varrho') \quad \Gamma \vdash e_0 : \tau_0 \quad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n \quad \tau'' <: \tau' \quad \tau <: \tau'}{\Gamma \vdash_{ref} \mathtt{new}\ e_0(e_1, \ldots, e_n) : \tau'/\bot}$$

$$\Gamma \vdash_{stm} \texttt{skip} \rhd \tau' \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash_{stm} e \rhd \tau'} \qquad \frac{\Gamma \vdash_{stm} s_1 \rhd \tau \quad \Gamma \vdash_{stm} s_2 \rhd \tau}{\Gamma \vdash_{stm} s_1 \texttt{;} s_2 \rhd \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash_{stm} \texttt{return } e \rhd \tau}$$

$$\frac{\Gamma \vdash e : \tau' \quad \Gamma \vdash_{stm} s_1 \rhd \tau \quad \Gamma \vdash_{stm} s_2 \rhd \tau}{\Gamma \vdash_{stm} \texttt{if (}e\texttt{) then \{}s_1\texttt{\} else \{}s_2\texttt{\}} \rhd \tau} \qquad \frac{\Gamma \vdash e : \tau' \quad \Gamma \vdash_{stm} s : \tau}{\Gamma \vdash_{stm} \texttt{while (}e\texttt{) \{}s\texttt{\}} \rhd \tau}$$

Fig. 4. Typing rules for statements

A call to a constructor function binds `this` to the new object. `this` of type τ'' is also the default return value if the function returns `undefined`. Otherwise, `new` returns whatever the constructor returns (τ). The subtyping built into the rule allows for both of these possibilities.

The rule for assignment infers the type of e_0 as a left-hand side.

$$\frac{\Gamma \vdash_{lhs} e_0 : \tau \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash_{ref} e_0 \texttt{ = } e_1 : \tau}$$

The typing judgment for left-hand sides has two rules, for variables and for property references. Property references are dealt with analogously to the rule for $\vdash_{ref}$, however, using the $\vdash_{upd}$ judgment in place of $\vdash_{acc}$. Since assignment *writes* to the reference, $\vdash_{upd}$ forces the type of the written value to be a lower bound for the type of the stored value. The judgment for $\vdash_{acc}$ behaves the other way round.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash_{lhs} x : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 \trianglerighteq \texttt{Obj}(\varphi_1)(\varrho_1) \quad \Gamma \vdash e_2 : \tau_2 \quad \vdash_{upd} \varrho_1 \mathbin{@} \tau_2 \hookleftarrow \tau'}{\Gamma \vdash_{lhs} e_1[e_2] : \tau'}$$

The typing rules for statements in Figure 4 are entirely standard. There is no subtyping rule for the return type of a statement because subtyping can be applied at the expression level before applying the rule for `return` .

5 Semantics

We specify a small-step operational semantics for Core JavaScript. To that end, we extend the syntax of expressions by store locations l drawn from a set *Loc*, variable references $var(l, str)$, property references $prop(l, str)$, and define one-step reduction relations for statements $\Sigma, l_0, s \rightarrow_s \Sigma', s'$ and expressions $\Sigma, l_0, e \rightarrow_e \Sigma', e'$, where Σ and $\Sigma' : Loc \rightarrow Storable$ are stores, $l_0 \in Loc$ is the address of an activation record, e, e' are expressions, and s, s' are statements. These relations are refined by address computation $\Sigma, l_0, e \rightarrow_a \Sigma', e$ and method access $\Sigma, l_0, e \rightarrow_m \Sigma', e$ which rely on further relations $\rightarrow_l$ (left-hand side of assignment) and $\rightarrow_v$ (variable access). Storables are defined by

$$\begin{aligned} Storable &= (String\ Constants + \{VALUE, \texttt{up}, \texttt{this}\}) \rightarrow Value \\ Value &= \{\texttt{undefined}, \texttt{null}\} + Boolean + Number + String + FValue + Loc \\ FValue &= \texttt{function}\ f@l(x_1, \ldots, x_n)\texttt{\{var}\ y_1, \ldots, y_m\texttt{;}\ s\texttt{\}} \end{aligned}$$

Expressions (see the appendix for the standard contextual rules)

$$\Sigma, l_0, \texttt{this} \to_e \Sigma, \Sigma(l_0)(\texttt{this})$$

$$\frac{\Sigma, l_0, x \to_v \Sigma', var(l, str)}{\Sigma, l_0, x \to_e \Sigma', \Sigma'(l)(str)}$$

$$\Sigma, l_0, var(l, str) = v \to_e \Sigma[l \mapsto \Sigma(l)[str \mapsto v]], v$$

$$\Sigma, l_0, prop(l, str) = v \to_e \Sigma[l \mapsto \Sigma(l)[str \mapsto v]], v$$

$$\Sigma, l_0, \{str_1 : v_1, \ldots, str_n : v_n\} \to_e \Sigma[l \mapsto [str_1 \mapsto v_1, \ldots, str_n \mapsto v_n]], l \quad \text{if } l \notin dom(\Sigma)$$

$$\Sigma, l_0, \texttt{function } f(x_1, \ldots, x_n)\{\texttt{var } y_1, \ldots, y_m; s\} \to_e \Sigma[l \mapsto [VALUE \mapsto \texttt{function } f@l_0(x_1, \ldots, x_n)\{\texttt{var } y_1, \ldots, y_m; s\}]], l \quad \text{if } l \notin dom(\Sigma)$$

$$\Sigma, l_0, l[str] \to_e \Sigma, \Sigma(l)(str)$$

$$\Sigma, l_0, l(v_1, \ldots, v_n) \to_e \Sigma[l' \mapsto [\texttt{this} \mapsto \texttt{null}, \texttt{up} \mapsto l_1, f \mapsto l, x_1 \mapsto v_1, \ldots, y_1 \mapsto \texttt{undefined}, \ldots]], \texttt{AR}(l', s) \quad \text{if } l' \notin dom(\Sigma), \Sigma(l)(VALUE) = \texttt{function } f@l_1(x_1, \ldots, x_n)\{\texttt{var } y_1, \ldots, y_m; s\}$$

$$\Sigma, l_0, prop(l'', str)(v_1, \ldots, v_n) \to_e \Sigma[l' \mapsto [\texttt{this} \mapsto l'', \texttt{up} \mapsto l_1, f \mapsto l, x_1 \mapsto v_1, \ldots, y_1 \mapsto \texttt{undefined}, \ldots]], \texttt{AR}(l', s) \quad \text{if } l' \notin dom(\Sigma), \Sigma(\Sigma(l'')(str))(VALUE) = \texttt{function } f@l_1(x_1, \ldots, x_n)\{\texttt{var } y_1, \ldots, y_m; s\}$$

$$\Sigma, l_0, \texttt{new } l(v_1, \ldots, v_n) \to_e \Sigma[l' \mapsto [\texttt{this} \mapsto l'', \texttt{up} \mapsto l_1, f \mapsto l, x_1 \mapsto v_1, \ldots, y_1 \mapsto \texttt{undefined}, \ldots], l'' \mapsto [\,]], \texttt{AR}(l', s) \quad \text{if } l', l'' \notin dom(\Sigma), \Sigma(l)(VALUE) = \texttt{function } f@l_1(x_1, \ldots, x_n)\{\texttt{var } y_1, \ldots, y_m; s\}$$

$$\Sigma, l_0, \texttt{AR}(l, \texttt{return } v) \to_e \Sigma, v$$

$$\Sigma, l_0, \texttt{AR}(l, \texttt{skip}) \to_e \Sigma, \texttt{undefined}$$

$$\frac{\Sigma, l, s \to_s \Sigma', s'}{\Sigma, l_0, \texttt{AR}(l, s) \to_e \Sigma', \texttt{AR}(l, s')}$$

$$\frac{\Sigma, l_0, e_0 \to_m \Sigma', e_0'}{\Sigma, l_0, e_0(e_1, \ldots) \to_e \Sigma', e_0'(e_1, \ldots)}$$

$$\frac{\Sigma, l_0, e_0 \to_l \Sigma', e_0'}{\Sigma, l_0, e_0 = e_1 \to_e \Sigma', e_0' = e_1}$$

Addresses (where $\to_l = \to_v \cup \to_a$ and $\to_m \supseteq \to_a$ where all cases omitted in $\to_a$ are identical to expression cases with $\to_e$ replaced by $\to_a$ in the conclusion)

$$\Sigma, l_0, x \to_v \Sigma, var(l_0, x) \quad \text{if } x \in dom(\Sigma(l_0))$$

$$\frac{\Sigma, \Sigma(l_0)(\texttt{up}), x \to_v \Sigma', v}{\Sigma, l_0, x \to_v \Sigma', v} \quad \text{if } x \notin dom(\Sigma(l_0)), \texttt{up} \in dom(\Sigma(l_0))$$

$$\Sigma, l_0, x \to_v \Sigma, var(l_0, x) \quad \text{if } x \notin dom(\Sigma(l_0)), \texttt{up} \notin dom(\Sigma(l_0))$$

$$\Sigma, l_0, l[str] \to_a \Sigma, prop(l, str) \qquad \frac{\Sigma, l_0, e_0 \to_e \Sigma', e_0'}{\Sigma, l_0, e_0[e_1] \to_a \Sigma', e_0'[e_1]} \qquad \frac{\Sigma, l_0, e_1 \to_e \Sigma', e_1'}{\Sigma, l_0, v_0[e_1] \to_a \Sigma', v_0[e_1']}$$

Fig. 5. Operational semantics

The elements of *Storable* are objects. They map property names to values. There are three special properties, *VALUE*, `up`, and `this`. The *VALUE* property is used by wrapper objects for primitive types and to store function closures. The `up` property is only used in environment objects that implement lexical scoping. It always points to the next lexically enclosing environment. The `this` property is reserved for the self reference in method calls and constructor functions. We leave the sets *Boolean*, *Number*, and *String* unspecified. Each element of the set *FValue* is a *function closure*. It registers the function name f for recursive use, the location l pointing to the lexically enclosing environment of the function's definition, the names $x_1, \ldots$ of the formal parameters, the names $y_1, \ldots$ of the local variables, and the statement s implementing the function's body.

Each of the reduction relations takes an initial store Σ, a reference to the current environment object l, and a syntactic object and rewrites it into the next state and a transformed object. To express the transformed syntactic objects requires to extend the syntax of expressions by values, which are the results of computations. They are generated by the grammar

$$
\begin{array}{rll}
v ::= & \texttt{undefined} & \\
| & \texttt{null} & \text{null reference} \\
| & c & \text{booleans, numbers, strings} \\
| & l & \text{location} \\
| & \mathit{var}(l, \mathit{str}) & \text{variable reference} \\
| & \mathit{prop}(l, \mathit{str}) & \text{property reference}
\end{array}
$$

The last two cases deserve some further explanation. A variable reference $\mathit{var}(l, \mathit{str})$ consists of the location l of an environment object and a property name (variable name) in that environment. Its main use is to serve as an address in the evaluation of an assignment expression. A property reference $\mathit{prop}(l, \mathit{str})$ consists of the location l of a program object and a property name. It serves as an address for evaluating assignments, but it also plays a role in detecting a method call.

A further new kind of expression is the activation object $\texttt{AR}(l, s)$. The location l is the address of an activation record and s is the statement to execute in this context. Each function application creates a new activation object initialized with the names and values of the parameters, the `this` pointer, the `up` pointer, and the local variables. On entry to an activation $\texttt{AR}(l, s)$, evaluation replaces the currently active context with the activation object l and proceeds with s.

Instead of explaining all the evaluation rules in detail, we concentrate on a few important details. Lookup for `this` only takes place in the toplevel activation object, every other variable is accessed through the scope chain (see definition of $\rightarrow_v$). As yet unknown variables are allocated in the toplevel activation object, which is identified by the absence of an `up` property.

Functions are objects that have a closure stored in the special *VALUE* property. The content of a closure is completely standard.

A function invocation can take three different forms: a function call, a method invocation, and a constructor call. These three forms differ solely in the way that `this` is determined, the rest of the activation object is constructed in the same way every time. In a function call, `this` is set to `null`[2]. For a method call, `this` is the receiving object and in a function called through a `new` expression, `this` is the newly constructed object. The last case is easily identified because its syntax is different. Distinguishing a method call from an ordinary function call requires to evaluate the function part to an address if possible (using the relation $\rightarrow_m$). If the result is a property reference, then we have a method call. Otherwise, it is an ordinary function call.[3]

[2] The standard prescribes that `this` should point to the toplevel activation object.
[3] We omit the `arguments` property to keep the presentation manageable.

6 Type Soundness

The connection between the semantics and the type system is made in the usual way by defining a notion of typed configurations and proving type preservation and progress for that notion. A configuration of Core JavaScript is a triple Σ, l_0, s of a store, a reference to an activation object, and a statement. There is also the auxiliary notion of an expression configuration Σ, l_0, e. The rewrite relations $\rightarrow_s$ and $\rightarrow_e$ of the operational semantics induce corresponding relations on configurations in the obvious way.

In addition to the type environment Γ for variables, there is a heap environment Δ for typing references in the store. It maps store locations to object types of the form $\mathtt{Obj}(\varphi)(\varrho)$. This type assignment must be compatible to the actual store: $\Delta \vdash_h \Sigma$. Compatibility means that $dom(\Delta) = dom(\Sigma)$ and that, for each $l \in dom(\Sigma)$ and $str \in dom(\Sigma(l))$, if $\Delta(l) = \mathtt{Obj}(\varphi)(str : \tau, \varrho)$ then the value $\Sigma(l)(str)$ has type τ. If any of the fields in φ is defined, then its contents describe $\Sigma(l)(\mathit{VALUE})$.

The typing rules for configurations define two judgments $\Delta, \Gamma \vdash_{sc} \Sigma, l_0, s \rhd \tau$ and $\Delta, \Gamma \vdash_{ec} \Sigma, l_0, e : \tau$, for statements and expressions. These judgments are defined analogously to $\vdash_{stm}$ and $\vdash$ but have additional rules for the new expressions introduced by the rewrite steps. Due to space reasons, we only give a few example rules.

$$\frac{\Delta(l) = \mathtt{Obj}(\varphi)(str : \tau, \varrho)}{\Delta, \Gamma \vdash_{ec} \Sigma, l_0, \mathit{var}(l, str) : \tau} \qquad \frac{\Delta, \mathit{TE}(\Delta, l) \vdash_{sc} \Sigma, l, s \rhd \tau}{\Delta, \Gamma \vdash_{ec} \Sigma, l_0, \mathtt{AR}(l, s) : \tau}$$

The second rule for an activation record is particularly interesting because it reconstructs a typing environment from the address l of the activation object of the function and from the heap type Δ. It relies on a function $\mathit{TE}(\Delta, l)$ that traverses the chain of activation objects beginning with l and constructs an environment from the properties (and their types) of the activation objects. We omit its straightforward specification.

Lemma 1 (Type preservation). *Suppose that $\Delta \vdash_h \Sigma$, $\Delta, \Gamma \vdash_{sc} \Sigma, l_0, s \rhd \tau$, and $\Sigma, l_0, s \rightarrow_s \Sigma', s'$.*

Then exists Δ' extending Δ such that $\Delta' \vdash_h \Sigma'$ and $\Delta', \Gamma \vdash_{sc} \Sigma', l_0, s' \rhd \tau$.

Lemma 2 (Progress). *Suppose that $\Delta \vdash_h \Sigma$, $\Delta, \Gamma \vdash_{sc} \Sigma, l_0, s \rhd \tau$.*

Then either $s =$ `skip`, $s =$ `return` v, or there exist Σ' and s' such that $\Sigma, l_0, s \rightarrow_s \Sigma', s'$.

7 Related Work

The main influence to this work is the work on soft typing and dynamic typing [3, 15, 6, 7]. These works define static type systems for dynamically typed languages, Scheme in these cases. The goal of these type systems is to enable compiler optimizations for Scheme programs. Dynamic typing has a twofold impact on the performance of a program. First, there is a memory overhead because each value

must carry with it a representation of its type. In the simplest implementation each value is boxed, that is, it is represented by a pointer to a heap-allocated cell. Clearly, there is also a time penalty for manipulating boxed values. Second, each operation must check that its arguments have the expected type before it can proceed to do the actual work. A soft typing system is able to identify the places where the dynamic type checks may be safely omitted and which values need not carry type information with them. The work by Henglein and Rehof [7] even specifies a translation into ML, a statically typed language that requires no type information at runtime.

Another group of works which is closely related to ours is the construction of type systems for the programming language Erlang [2]. Erlang poses similar problems as JavaScript, but is simpler in some respects. For example, functions have fixed arity and there are no structures comparable with JavaScript's objects. Two type systems have been constructed for Erlang, one based on standard type theory [8] and another one which appears more ad-hoc [10]. Both system work from programmer specified type signatures, whereas our system is targeted towards performing automatic program analysis.

The rows in our object types are clearly inspired by type systems for records [13]. The main difference to a traditional record system is the use of strings as labels, which requires a first-class treatment of labels [5, 9] but in the guise of singleton types.

8 Conclusion

We have presented a first attempt at defining a type system for analyzing a weakly typed scripting language, JavaScript. The system is guided by a matching relation which specifies type convertibility. The matching relation determines how conservative the system is and which conversions should only be flagged instead of being rejected (*e.g.*, converting `null` to an object).

A number of extensions might be considered: the object prototyping mechanism, more general type indices, and polymorphism. However, practical experience is necessary to select the most urgently needed one. An implementation is under way to evaluate the type system with typical JavaScript programs.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, NY, 1993.
3. Robert Cartwright and Mike Fagan. Soft typing. In *Proc. Conference on Programming Language Design and Implementation '91*, pages 278–292, Toronto, Canada, June 1991. ACM.
4. ECMAScript Language Specification. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, December 1999. ECMA International, ECMA-262, 3rd edition.

5. Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Dept. of Computer Science, University of Nottingham, November 1996.
6. Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22:197–230, 1994.
7. Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In Simon Peyton Jones, editor, *Proc. Functional Programming Languages and Computer Architecture 1995*, La Jolla, CA, June 1995. ACM Press, New York.
8. Simon Marlow and Philip Wadler. A practical subtyping system for erlang. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 136–149. ACM Press, 1997.
9. Susumu Nishimura. Static typing for dynamic messages. In Luca Cardelli, editor, *Proc. 25th Annual ACM Symposium on Principles of Programming Languages*, pages 266–278, San Diego, CA, USA, January 1998. ACM Press.
10. Sven-Olof Nyström. A soft-typing system for erlang. In *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 56–71. ACM Press, 2003.
11. The Mozilla Organization. Rhino: JavaScript for Java. `http://www.mozilla.org/rhino/`, September 2004.
12. François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 276–290. ACM Press, 1999.
13. Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1994.
14. David Ungar and Randall B. Smith. SELF: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–206, July 1991.
15. Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1):87–152, January 1997.
16. Kwangkeun Yi. An abstract interpretation for estimating uncaught exceptions in standard ML programs. *Science of Computer Programming*, 31(1):147–173, 1998.

Java Jr.: Fully Abstract Trace Semantics for a Core Java Language

Alan Jeffrey[1,2,⋆] and Julian Rathke[3]

[1] Bell Labs, Lucent Technologies, Chicago, IL, USA
[2] DePaul University, Chicago, IL, USA
[3] University of Sussex, Brighton, UK

Abstract. We introduce an expressive yet semantically clean core Java-like language, Java Jr., and provide it with a formal operational semantics based on traces of observable actions which represent interaction across package boundaries. A detailed example based on the Observer Pattern is used to demonstrate the intuitive character of the semantic model. We also show that our semantic trace equivalence is fully-abstract with respect to a natural notion of testing equivalence for object systems. This is the first such result for a full class-based OO-language with inheritance.

1 Introduction

Operational semantics as a modelling tool for program behaviour originated in the early 1960s in early work of McCarthy [19] and found some popularity in modelling programming languages such as ALGOL and LISP [20, 30, 13] and the lambda-calculus, [18]. Later, this approach to modelling was championed by Plotkin [25, 26] and has since been applied extensively and successfully for providing semantic descriptions of simple programming languages and computational models [31, 21, 22, 15, 29, 1, 11, 8]. As these modelling techniques began to be applied to to larger scale languages, their semantic descriptions became more complex [27, 28, 4, 9, 6, 3].

There has been a considerable research effort towards formalising operational behaviour of Java and Java-like languages, for example [4, 11, 16, 6, 9, 24, 3]. Indeed [3] is a special volume journal dedicated to semantic techniques for the Java language which collects together much of the interesting work on this topic to date. None of these, however, address the issue of program equivalence and extensional descriptions of object behaviour. The papers cited above tend to analyse subsets of the Java language for issues related to type safety rather than equivalence. Banerjee and Naumann [5] provide a denotational semantics for a subset of Java, but do not prove a correspondence with an operational model. With this in mind we propose an experimental class-based Java-like language, designed to have a straightforward semantic description of the interactive behaviour of its programs. We call this language Java Jr.

⋆ This material is based upon work supported by the National Science Foundation under Grant No. 0430175.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 423–438, 2005.

To address the issue of program equivalence in Java Jr. we make use of Morris' theory of testing [23], later refined by Hennessy [15]. It is a robust theory based on observability of basic events during computation in context and has been applied to many languages, including models of functional programming such as the λ-calculus [2] and PCF [25], concurrent languages such as the π-calculus [10], and object-oriented languages such as the σ-calculus [17].

The definition of testing equivalence involves a universal quantification over all possible test harnesses for programs, which often makes establishing equivalences difficult. For this reason it is commonplace to investigate alternative characterisations of semantic equivalence which offer simpler proof techniques. We provide an alternative characterisation of testing equivalence in Java Jr. by describing sequences of interactions which programs may engage in with arbitrary test harnesses. These are defined as traces derived from a labelled transition system [7]. The key result we prove is that programs which exhibit the same set of traces are exactly those which are testing equivalent. This property of our trace model is known as full abstraction.

For the remainder of the paper, we will present an overview of the Java Jr. language followed by its formal syntax and operational model. We will then discuss issues of typeability and how well-typed object components can grouped together to form larger, well-typed systems. In Section 4, we define our notion of testing equivalence for Java Jr. and in the following section introduce the trace model. The full abstraction result is outlined in Section 6 and we then close with remarks about future work.

2 The Java Jr. Language

Java Jr. is a small, single threaded, subset of the Java language which allows for the declaration of classes and interfaces in packages. It includes two extensions of Java: it allows for packages to contain object declarations (rather than requiring them to be static fields inside classes), and it allows for explicit specification of the signature of a package. We shall discuss these in more detail below.

An example Java Jr. program is given in Figure 1: it provides a simple implementation of the *Observer* pattern from [12]. Observer objects can register themselves with a Subject (in this case, we just provide a singleton Subject), and any calls to `notify` on the Subject result in `update` calls on all the registered Observers.

We will use the following terminology throughout this paper: a *package* consists of a sequence of *declarations* and a *component* consists of a sequence of packages. We use the metavariables C, P and D to represent components, packages and declarations respectively. We will also use the overbar notation to denote sequences, for example $\bar{P}$ refers to a sequence of packages. The metavariable v is used throughout the paper to refer to a fully specified object reference including the package name and object identity, using the usual Java $p.o$ syntax. We also use the metavariable t to refer to types of the language, that is, fully specified class or interface names. For example, in Figure 1, `observer.singleton` is a fully specified object reference, and `observer.Subject` is a fully specified interface name.

The notion of packages are central to Java Jr. They delimit our semantic descriptions by identifying the boundaries of observable interactions. Statically, only interfaces and

```
{ package observer;
  interface Subject extends ε {
    System.void addObserver (observer.Observer o);
    System.void notify ();
  }
  interface Observer extends ε {
    System.void update ();
  }
  class SubjectImpl extends Object implements Subject {
    observer.List contents;
    SubjectImpl (observer.List contents) { super (); this.contents = contents; }
    public System.void addObserver (observer.Observer o) {
      return (this.contents = new observer.Cons (o, this.contents), System.unit);
    }
    public System.void notify () { return this.contents.updateAll (); }
  }
  class List extends Object implements ε {
    observer.List () { super (); }
    public System.void updateAll () { return System.unit; }
  }
  class Cons extends List {
    observer.Observer hd; observer.List tl;
    Cons (observer.Observer hd, observer.List tl) { super (); this.hd = hd; this.tl = tl; }
    public System.void updateAll () { return (this.hd.update(), this.tl.updateAll ()); }
  }
  object observer.SubjectImpl singleton implements observer.Subject {
    contents = observer.list_nil;
  }
  object observer.List list_nil implements ε { }
}
```

Fig. 1. Definition of the `observer` package in Java Jr.

public objects (and not classes or private objects) are visible across package boundaries; dynamically, only publicly visible method calls (and not fields, constructors, or private methods) are visible across package boundaries. In particular, code placed in a package p, cannot create instances of objects using classes in a different package q. Nor can this code access fields of objects created in q directly. In line with software engineering good practice, each of these operations must be provided by factory, accessor and mutator methods. Moreover, all packages in Java Jr. are *sealed*, that is new classes, objects and interfaces may not be added to existing packages.

Where Java Jr. differs significantly from Java is in the provision for statically available methods and members. Rather than modelling the intricacies of Java's `static` modifier, we allow packages to contain explicit object declarations of the form:

$$\mathsf{object}\ t\ o\ \mathsf{implements}\ \bar{t}\{f_1 = v_1; , \ldots , f_n = v_n;\}$$

Such a declaration indicates that an object with identity o is an instance of class t with initial field assignments $f_i = v_i$; which may change during program execution. Object declarations also contain a list of interface types $\bar{t}$ which the object is said to *implement*. These are the externally visible types for the object, as opposed to the class name t, which is only internally visible within the package. If the list of interface types is empty, then the object is considered private to the package. For example, in Figure 1 we have:

```
{ package observer;
  interface Subject extends ε {
    System.void addObserver (observer.Observer o);
    System.void notify ();
  }
  interface Observer {
    System.void update ();
  }
  extern observer.Subject singleton;
}
```

Fig. 2. External view of of the `observer` package

- Object `singleton` is declared as having class `SubjectImpl`, and implementing `Subject`, so within the `observer` package we have `singleton:SubjectImpl` but externally we only have `singleton:Subject`.
- Object `list_nil` implements no interfaces, so within the `observer` package we have `list_nil:List` but externally it is inaccessible.

In Java, all packages are *export* packages, that is they contain both the signature of the package and its implementation: in contrast, languages like C allow for importation of externally defined entities, and for the importer to give the signature of the imported entity. In defining a notion of equivalence for Java programs, we found it necessary to be formal about the notion of package interface, since the external behaviour of a package crucially depends on the types of external entities.

For this reason, our other extension of Java is to allow for *import* packages, which do not contain class or object declarations, and instead only contain interface declarations and *extern* declarations, of the form:

$$\mathsf{extern}\ \bar{t}\ o;$$

Such an declaration within an import package p declares that any export package which implements p must provide an object named o with public types $\bar{t}$. For example, in Figure 2 we give the *external view* of the `observer` package.

2.1 Formal Syntax and Semantics of Java Jr.

We present a formal grammar for the Java Jr. language in Figure 3. For the most part this syntax is imported directly from Java.

The only novel Java Jr. expression is of the form $E\ \mathsf{in}\ p$ which has no effect upon runtime behaviour but is used simply as an annotation to assist typechecking. This operator is effectively a type coercion of the following form:

> If the expression E is well-typed to run in package p with return type t, then the expression $E\ \mathsf{in}\ p$ is well-typed to run in any package q with return type t, as long as t is a visible type in q.

In order to present the dynamic and static semantics of our language we found it useful to make recourse to a number of auxiliary, syntactically defined functions. The definitions of these are largely obvious and are too numerous to list here. One of the most important

Components:	$C ::= \bar{P}$
Packages:	$P ::= \{\mathsf{package}\ p; \bar{D}\}$
Declarations:	$D ::= \mathsf{class}\ c\ \mathsf{extends}\ t\ \mathsf{implements}\ \bar{t}\,\{K\,\bar{G}\,\bar{M}\}$
	$\mid\ \mathsf{interface}\ i\ \mathsf{extends}\ \bar{t}\,\{\bar{N}\}$
	$\mid\ \mathsf{object}\ t\ o\ \mathsf{implements}\ \bar{t}\,\{\bar{F}\}$
	$\mid\ \mathsf{extern}\ \bar{t}\ o; \quad (\bar{t} \neq \varepsilon)$
Constructors:	$K ::= c\,(\bar{t}\ \bar{f}, \bar{u}\ \bar{g})\,\{\mathsf{super}\,(\bar{f}); \mathsf{this}.\bar{g} = \bar{g};\}$
Fields:	$F ::= f = v;$
Field types:	$G ::= t\ f;$
Methods:	$M ::= \mathsf{public}\ t\ m\,(\bar{t}\ \bar{x})\,\{\mathsf{return}\ E;\}$
Method types:	$N ::= t\ m\,(\bar{t}\ \bar{x});$
Expressions:	$E ::= v \mid x \mid E.m(\bar{E}) \mid E.f \mid E.f = E$
	$\mid\ \mathsf{new}\ t\,(\bar{E}) \mid (E == E\,?\,E : E) \mid E, E \mid E\ \mathsf{in}\ p$
Compound names:	$p, \ldots, w ::= \bar{a}$

Simple names range over Object, and $a, \ldots, o$ and Variables range over this and $x, \ldots, z$. We also assume that sequences of field identifiers and variables, $\bar{f}$ and $\bar{x}$, and names in $\bar{P}, \bar{D}, \bar{F}, \bar{G}, \bar{M}, \bar{N}$ are always pairwise distinct.

Fig. 3. Syntax of the Java Jr. language

of these is the updating function $C + C'$ which is an asymmetric operator in which each declaration $\{\mathsf{package}\ p; D\}$ within C' overrides any declaration with the same full name present in C, is included in package p of C if C contains this package, and is simply appended to C otherwise. We write $C.p.n$ for the declaration $\{\mathsf{package}\ p; D\}$ where package p in C declares D with name n. Another crucial definition is

- $C.p$ is an *export package* if there is a n such that $C.p.n = \{\mathsf{package}\ p; D\}$ where D is either a class or an object declaration.
- $C.p$ is an *import package* if it is not an export package.

2.2 Dynamic Semantics

A Java Jr. component C, will exhibit no dynamic behaviour until a thread of execution is provided. As Java Jr. is a single threaded language we need not concern ourselves with thread identities and synchronisation and we may model the single active thread simply by a Java Jr. expression E. Given this, it is not difficult to define a relation $\rightarrow$ of the form

$$(C \vdash E) \rightarrow (C' \vdash E')$$

to model the evaluation of the thread E with respect to the component C. In order to define the reduction relation it is useful to identify what is typically referred to as *evaluation contexts* [32]. The grammar of all possible evaluation contexts of the language is given by

$$\begin{aligned}\mathcal{E} ::= {} & \cdot \mid \mathcal{E}.m(\bar{E}) \mid v.m(\bar{v}, \mathcal{E}, \bar{E}) \mid \mathcal{E}.f \mid \mathcal{E}.f = E \mid v.f = \mathcal{E} \mid \mathsf{new}\ t\,(\bar{v}, \mathcal{E}, \bar{E}) \\ & \mid (\mathcal{E} == E\,?\,E_T : E_F) \mid (v == \mathcal{E}\,?\,E_T : E_F) \mid \mathcal{E}, E \mid \mathcal{E}\ \mathsf{in}\ p\end{aligned}$$

We also list, in Figure 4, the proof rules which define the reduction relation itself. For the most part, these rules are reasonably straightforward. Two points of interest are:

$$\frac{C.v = \{\mathsf{package}\ p; \mathsf{object}\ t\ o\ \mathsf{implements}\ \bar{t}\,\{\bar{F}\}\} \quad \mathsf{public}\ u\ m\,(\bar{u}\ \bar{x})\,\{\mathsf{return}\ E;\} \in C.t.\mathsf{methods}}{(C \vdash \mathcal{E}[v.m\,(\bar{v})]) \to (C \vdash \mathcal{E}[E[v/\mathsf{this}, \bar{v}/\bar{x}]\ \mathsf{in}\ p])}$$

$$\frac{C.v = \{\mathsf{package}\ p; \mathsf{object}\ t\ o\ \mathsf{implements}\ \bar{t}\,\{\bar{F}\}\} \quad f = w; \in \bar{F}}{(C \vdash \mathcal{E}[v.f]) \to (C \vdash \mathcal{E}[w])}$$

$$\frac{C.v = \{\mathsf{package}\ p; \mathsf{object}\ t\ o\ \mathsf{implements}\ \bar{t}\,\{\bar{F}\}\} \quad (f = u;) \in \bar{F} \quad C' = C + \{\mathsf{package}\ p; \mathsf{object}\ t\ o\ \mathsf{implements}\ \bar{t}\,\{\bar{F}'\}\} \quad \bar{F}' = \bar{F} + (f = w;)}{(C \vdash \mathcal{E}[v.f = w]) \to (C' \vdash \mathcal{E}[w])}$$

$$\frac{C.p.c.\mathsf{fields} = \bar{t}\ \bar{f}; \quad p.o \notin \mathsf{dom}\,(C) \quad C' = C + \{\mathsf{package}\ p; \mathsf{object}\ p.c\ o\ \mathsf{implements}\ \varepsilon\,\{\bar{f} = \bar{v};\}\}}{(C \vdash \mathcal{E}[\mathsf{new}\ p.c\,(\bar{v})]) \to (C' \vdash \mathcal{E}[p.o])} \qquad \frac{}{(C \vdash \mathcal{E}[v\ \mathsf{in}\ p]) \to (C \vdash \mathcal{E}[v])}$$

$$\frac{}{(C \vdash \mathcal{E}[(v == v\,?\,E : E')]) \to (C \vdash \mathcal{E}[E])} \qquad \frac{v \neq w}{(C \vdash \mathcal{E}[(v == w\,?\,E : E')]) \to (C \vdash \mathcal{E}[E'])}$$

Fig. 4. Rules for reductions $(C \vdash E) \to (C' \vdash E')$

- In the rule for generating new objects, the new object is always stored within the same package as the class it is instantiating.
- The result of a method call is to inline the method body E, say, within the current evaluation context. Note that before doing this E is wrapped with the coercion $E\ \mathsf{in}\ p$ where p is the package of the receiver. This facilitates type-safe embedding of external code within a package at runtime.

Note that the statically defined component C is modified during reduction as it also models the runtime heap as well as the program class table.

2.3 Static Semantics

As with Java itself, Java Jr. is a statically typed class-based language. It uses the package mechanism to enforce *visibility*: in Java Jr., classes are always package protected, and interfaces are always public, conforming to the common discipline of *programming to an interface*. In order to check that a Java Jr. program respects package visibility, the type system tracks the current package of each class, method and expression, for example the type judgement for an expression is:

$$C \vdash E : t\ \mathsf{in}\ p$$

This indicates that the expression E could potentially access all protected fields and methods in p but cannot access anything outside of p except public methods declared in interfaces.

We close this section by confirming that Java Jr. satisfies Subject Reduction for the runtime type system.

Proposition 1 (Subject Reduction). *For any well-typed component* $\vdash C : \mathsf{component}$ *such that* $C \vdash E : t$ in p *and* $(C \vdash E) \rightarrow^* (C' \vdash E')$ *we have that* $\vdash C' : \mathsf{component}$ *and* $C' \vdash E' : t$ in p.

3 Linking and Compatibility

A fundamental property of components ought to be that they should be compositional: it should be possible to replace a subcomponent with an an equivalent subcomponent without affecting the whole system. In Section 4 we will discuss the dynamic properties of equivalence, and in this section we will discuss the static properties. Our goal is to provide a characterisation for when we can replace a subcomponent of a well-typed system and ensure that the new system is still well-typed.

When first need to discuss what *linking* means in the context of Java Jr. Consider two components C_1, which contains an import package p, and C_2, which contains an export package p. As long as C_1 and C_2 are *linkable*, we should be able to find a component $C_1 \between C_2$ where C_1's import of p is satisfied by C_2's export.

We can now define when it is possible to link two declarations. Declarations D_1 and D_2 are *linkable* if one of the following cases holds:

- D_1 is object $t\ o$ implements $\bar{t}\{\bar{F}\}$ and D_2 is extern $\bar{t}\ o$;
- D_2 is object $t\ o$ implements $\bar{t}\{\bar{F}\}$ and D_1 is extern $\bar{t}\ o$;
- $D_1 = D_2$ and are interface or extern declarations.

We define when it is possible to link two packages of the same name. Given packages P_1 and P_2 we say that these are *linkable* if one of the following cases holds:

- P_1 is an export package and P_2 is an import package, and for each v such that $P_2.v = \{\mathsf{package}\ p; D_2\}$ we have that $P_1.v = \{\mathsf{package}\ p; D_1\}$ where D_1 and D_2 are linkable.
- Symmetrically, when P_1 is an import package and P_2 is an export package.
- P_1 and P_2 are both import packages, and for each v such that $P_1.v = \{\mathsf{package}\ p; D_1\}$ and $P_2.v = \{\mathsf{package}\ p; D_2\}$ we have that $D_1 = D_2$.

We define when it is possible to link two components: C_1 and C_2 are *linkable* if

- for any $P_1 \in C_1$ and $P_2 \in C_2$ with $\mathsf{name}(P_1) = \mathsf{name}(P_2)$ we have that P_1 and P_2 are linkable.

The above definitions outline the formal requirements for two components C_1 and C_2 to be linked to form the larger component $C_1 \between C_2$ given by:

$$C_1 \between C_2 = (C_1.\mathsf{imports} + C_2.\mathsf{imports}) + (C_1.\mathsf{exports} + C_2.\mathsf{exports})$$

where $C.\mathsf{exports}$ is the component containing all of the export packages of C, and similarly for $C.\mathsf{imports}$.

Proposition 2. *If* $\vdash C_1 : \mathsf{component}$ *and* $\vdash C_2 : \mathsf{component}$ *and* C_1 *and* C_2 *are linkable then* $\vdash C_1 \between C_2 : \mathsf{component}$.

We can now address our goal of providing a characterisation for when we can replace a subcomponent C_1 of a well-typed system $C_1 \between C$ by a replacement component C_2 and be sure that $C_2 \between C$ is still well-typed. We shall call such components C_1 and C_2 *compatible*, defined as:

$$\begin{array}{c}\text{for all } C,\ C \text{ and } C_1 \text{ are linkable}\\ \text{if and only if } C \text{ and } C_2 \text{ are linkable}\end{array}$$

This definition, although appealing for its intuitive character, may be a little intractable due to the use of the quantification over all components C. For this reason we seek to provide a direct syntactic characterisation of compatibility. Two components C_1 and C_2 are *interface compatible* when:

$$\begin{array}{c}\text{for all } t,\ C_1.t = \{\mathsf{package}\ p; \mathsf{interface}\ i\ \mathsf{extends}\ \bar{t}\,\{\bar{N}\}\}\\ \text{if and only if } C_2.t = \{\mathsf{package}\ p; \mathsf{interface}\ i\ \mathsf{extends}\ \bar{t}\,\{\bar{N}\}\}\end{array}$$

Two components C_1 and C_2 are *extern compatible* when:

$$\begin{array}{c}\text{for all } v,\ C_1.v = \{\mathsf{package}\ p; \mathsf{extern}\ \bar{t}\ o;\}\\ \text{if and only if } C_2.v = \{\mathsf{package}\ p; \mathsf{extern}\ \bar{t}\ o;\}\end{array}$$

Two components C_1 and C_2 are *object compatible* when:

$$\begin{array}{c}\text{for all } v \text{ and } \bar{t} \neq \varepsilon,\ C_1.v = \{\mathsf{package}\ p; \mathsf{object}\ t_1\ o\ \mathsf{implements}\ \bar{t}\,\{\bar{F}_1\}\}\\ \text{if and only if } C_2.v = \{\mathsf{package}\ p; \mathsf{object}\ t_2\ o\ \mathsf{implements}\ \bar{t}\,\{\bar{F}_2\}\}\end{array}$$

Two components C_1 and C_2 are *package compatible* when:

$$\begin{array}{c}\text{for all } p,\ p \in \mathsf{dom}\,(C_1) \text{ and } C_1.p \text{ is an export package}\\ \text{if and only if } p \in \mathsf{dom}\,(C_2) \text{ and } C_2.p \text{ is an export package}\end{array}$$

For readers familiar with Java's notion of *binary compatibility* [14–Chapter 13], these are stronger requirements, justified by the following result.

Proposition 3. *Components* $\vdash C_1 : \mathsf{component}$ *and* $\vdash C_2 : \mathsf{component}$ *are compatible if and only if they are interface, extern, object and package compatible.*

4 Contextual Equivalence

The question of whether two programs are equal lies at the heart of semantics. An initial requirement for equivalence clearly should be that the programs, or components, are at least compatible. Further to this, we adopt an established means of defining equivalence by making use of contextual testing [15, 23]; programs are considered equal when they pass exactly the same tests.

In the case of Java Jr., a test is any component which can be linked against the component being tested, and the resulting system passes a test by printing an appropriate message using a chosen method `System.out.print(Object)`. The remainder of this section will now formalise this notion of testing.

Define a special component System as:

```
{ package System;
  interface Output { Object print(Object msg); }
  extern System.Output out;
}
```

We say that a component C *accepts* System if C and System are linkable and C.`System` is not an export package. Note that if $\texttt{System} \notin \mathsf{dom}(C)$ then C trivially accepts System. For compatible components $\vdash C_1 : \mathsf{component}$ and $\vdash C_2 : \mathsf{component}$ which accept System, define $C_1 \lesssim C_2$ as:

for all $\vdash C : \mathsf{component}$ linkable with C_1 and $C \between C_1 \vdash E : \mathsf{Object\ in} * $ and for all $C \vdash v : \mathsf{Object\ in} *$ we have

$$(C \between C_1 \vdash E) \rightarrow^* (C_1' \vdash \mathcal{E}_1[\texttt{System.out.print}(v)]) \quad \text{implies}$$
$$(C \between C_2 \vdash E) \rightarrow^* (C_2' \vdash \mathcal{E}_2[\texttt{System.out.print}(v)])$$

We say that well-typed C_1 and C_2 are contextually equivalent, $C_1 \simeq C_2$ whenever both

$$C_1 \lesssim C_2 \qquad \text{and} \qquad C_2 \lesssim C_1 .$$

Although this definition is appealing in the sense of being extensional and robust, it is rather intractable as a means of identifying equivalent programs, due to the quantification over all well-typed components C. We will now establish a simpler trace-based method for establishing contextual equivalence for Java Jr.

5 Trace Semantics

We will now discuss the trace semantics of Java Jr., which provides a description of the external behaviour of a component as a series of method calls and returns. The semantics of a component describes all possible interactions it could engage in with some unknown testing component. Each interaction takes the form of a sequence of basic actions α given by:

$$\gamma ::= v.m(\bar{v}) \mid \mathsf{return}\ v \mid \mathsf{new}(v) . \gamma$$
$$a ::= \gamma? \mid \gamma! \qquad \alpha ::= a \mid \tau$$

Each visible action is either a method call $v.m(\bar{v})$ or method return v. They are decorated $\gamma?$ if the message goes from the environment to the process, or $\gamma!$ if the message comes from the process to the environment. Moreover, actions may mention new objects which have not previously been seen: these are indicated by $\mathsf{new}(v) . \gamma$. The final action τ is used to represent interaction internal to the component under test.

We define *traces* as sequences $\bar{a}$ of visible actions, considered up to *alpha equivalence*, viewing $\mathsf{new}(v) . \bar{a}$ as a binder of v in $\bar{a}$:

$$\bar{a} . \mathsf{new}(\bar{v}) . \bar{b} \equiv \bar{a} . \mathsf{new}(\bar{w}) . \bar{b}[\bar{w}/\bar{v}] \quad \text{when } \bar{w} \notin \bar{b}$$

We will now describe the rules which generate traces from the component syntax. Before we can do this though it is useful to present an auxiliary notion.

The downcasting of imported names $C + \mathsf{extern}\ t\ v;$ is given by:

$$\begin{aligned} C + \mathsf{extern}\ t\ v; &= C \quad (\text{when } C \vdash v : t \text{ in } *) \\ C + \mathsf{extern}\ t\ v; &= C + \{\mathsf{package}\ p; \mathsf{extern}\ \bar{t}, t\ o;\} \\ &\qquad (\text{otherwise, where } C.v = \{\mathsf{package}\ p; \mathsf{extern}\ \bar{t}\ o;\} \\ &\qquad \text{and } C.\bar{t}.\mathsf{headers} \cup C.t.\mathsf{headers} \text{ are compatible}) \end{aligned}$$

Similarly, the downcasting of exported names $C + \mathsf{object}\ t\ v;$ is given by:

$$\begin{aligned} C + \mathsf{object}\ t\ v; &= C \quad (\text{when } C \vdash v : t \text{ in } *) \\ C + \mathsf{object}\ t\ v; &= C + \{\mathsf{package}\ p; \mathsf{object}\ u\ o\ \mathsf{implements}\ \bar{t}, t\ \{\bar{F}\}\} \\ &\qquad (\text{otherwise, when } C.v = \{\mathsf{package}\ p; \mathsf{object}\ u\ o\ \mathsf{implements}\ \bar{t}\ \{\bar{F}\}\} \\ &\qquad \text{and } C \vdash u <: t \text{ in } p \text{ and } \bar{t} \neq \varepsilon) \end{aligned}$$

Notice that downcasting with $t\ v$ has no effect in case the object reference v is already known to the component at the (public) type t. Otherwise, the appropriate import or export declaration is updated.

In order to generate traces we need to describe all possible interactions of components with an unknown testing component and unknown thread. We build these interactions up from sequences of single basic actions which the component can engage in. There are essentially two modes of interaction we need to consider here. One is the situation in which the unknown testing component and thread is executing in its code and may call in to a method of the component under test. The other is the situation in which the component under test has been called and is executing some of its known code. We represent these two scenarios using the following *states*:

$$\Sigma ::= (C \vdash E : t \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \mid (C \vdash \mathsf{block} \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u})$$

where block represents unknown code being executed by the testing environment and $\bar{\mathcal{E}}$ represents the component C's view of the evaluation stack. In fact this stack is formed from a *sequence* of evaluation contexts as the view of the full evaluation stack is only partial. The types of these evaluation contexts is also recorded and uses the notation $\mathcal{E} : t \to u$ to indicate that the hole in $\mathcal{E}$ is to be filled with an expression of type t, and doing so will yield an expression of type u.

We now define a relation $\Sigma \stackrel{\bar{b}}{\Longrightarrow} \Sigma'$ between (well-typed) states which describes the sequences of actions a component can engage in. The defining rules for this relation are presented in Figure 5. From here we are now in a position to define the semantics of a component as

$$\mathsf{Traces}(C) = \{\bar{a} \mid (C \vdash \mathsf{block} \rhd \varepsilon : \varepsilon) \stackrel{\bar{b}}{\Longrightarrow} \Sigma \text{ and } \bar{a} \equiv \bar{b}\}$$

In Figure 6, we show an example of our Observer example above. We define a component Test which contains (an external declaration of) an object which will be registered with the observer service:

```
{ package observer.test;
  interface Test { void run (); }
  extern observer.test.Test test;
}
```

$$\frac{(C \vdash E) \to (C' \vdash E')}{(C \vdash E : t \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{\tau} (C' \vdash E' : t \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u})}$$

Silent transitions

$$\frac{C.v \text{ is an export} \quad C \vdash v : u \text{ in } * \quad s\, m\,(\bar{s}\,\bar{x}); \in C.u.\mathsf{headers} \quad C' = C + \mathsf{extern}\ \bar{s}\ \bar{v};}{(C \vdash \mathsf{block} \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{v.m\,(\bar{v})?} (C' \vdash v.m\,(\bar{v}) : s \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u})}$$

$$\frac{C' = C + \mathsf{extern}\ t\ v;}{(C \vdash \mathsf{block} \rhd \mathcal{E}, \bar{\mathcal{E}} : t \to u, \bar{t} \to \bar{u}) \xrightarrow{\mathsf{return}\ v?} (C' \vdash \mathcal{E}[v] : u \rhd \bar{\mathcal{E}} : \bar{t})}$$

Input transitions

$$\frac{C.v \text{ is an import} \quad C \vdash v : u \text{ in } * \quad s\, m\,(\bar{s}\,\bar{x}); \in C.u.\mathsf{headers} \quad C' = C + \mathsf{object}\ \bar{s}\ \bar{v};}{(C \vdash \mathcal{E}[v.m\,(\bar{v})] : t \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{v.m\,(\bar{v})!} (C' \vdash \mathsf{block} \rhd \mathcal{E}, \bar{\mathcal{E}} : s \to t, \bar{t} \to \bar{u})}$$

$$\frac{C' = C + \mathsf{object}\ t\ v;}{(C \vdash v : t \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{\mathsf{return}\ v!} (C' \vdash \mathsf{block} \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u})}$$

Output transitions

$$\frac{\begin{array}{c} C.p \text{ is an import package} \quad p.o \notin \mathsf{dom}\,(C) \quad p.o \in \mathsf{fn}\,(\gamma?) \\ C'' = C + \{\mathsf{package}\ p; \mathsf{extern}\ \mathsf{Object}\ o;\} \\ (C'' \vdash \mathsf{block} \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{\gamma?} (C' \vdash E' : t' \rhd \bar{\mathcal{E}}' : \bar{t}' \to \bar{u}') \end{array}}{(C \vdash \mathsf{block} \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{\mathsf{new}(p.o).\gamma?} (C' \vdash E' : t' \rhd \bar{\mathcal{E}}' : \bar{t}' \to \bar{u}')}$$

$$\frac{\begin{array}{c} C.p.o = \{\mathsf{package}\ p; \mathsf{object}\ u\ o\ \mathsf{implements}\ \varepsilon\,\{\bar{F}\}\} \quad p.o \in \mathsf{fn}\,(\gamma!) \\ C'' = C + \{\mathsf{package}\ p; \mathsf{object}\ u\ o\ \mathsf{implements}\ \mathsf{Object}\,\{\bar{F}\}\} \\ (C'' \vdash E : t \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{\gamma!} (C' \vdash \mathsf{block} \rhd \bar{\mathcal{E}}' : \bar{t}' \to \bar{u}') \end{array}}{(C \vdash E : t \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{\mathsf{new}(p.o).\gamma!} (C' \vdash \mathsf{block} \rhd \bar{\mathcal{E}}' : \bar{t}' \to \bar{u}')}$$

Fresh name transitions

$$\frac{}{\Sigma \xRightarrow{\varepsilon} \Sigma} \qquad \frac{\Sigma \xRightarrow{\bar{a}} \Sigma' \xRightarrow{\bar{a}'} \Sigma''}{\Sigma \xRightarrow{\bar{a}\,\bar{a}'} \Sigma''} \qquad \frac{\Sigma \xrightarrow{\tau} \Sigma'}{\Sigma \xRightarrow{\varepsilon} \Sigma'} \qquad \frac{\Sigma \xrightarrow{a} \Sigma'}{\Sigma \xRightarrow{a} \Sigma'}$$

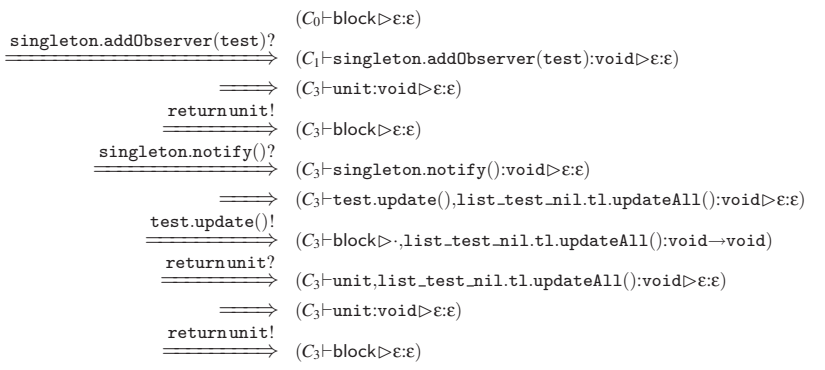
Concatenating actions

Fig. 5. Generating rules for labelled transitions

Note that during this example the type of the `test` object changes: initially it is just `observer.test.Test`, but after the first action it also has type `observer.Observer`.

6 Full Abstraction

Having built our trace model of components we now need to verify that the notion of equivalence induced by the model (equality on trace sets), does actually coincide with the intuitive notion of testing equivalence $\simeq$ defined earlier. This is the content of the Full Abstraction Theorem.



C_0 = Components System, Test and that defined in Figure 1.
$C_1 = C_0 +$ {package observer.test; extern Test, Observer test; }
$C_2 = C_1 +$ {package observer; object Cons list_test_nil { hd=test; tl=list_nil; }}
$C_3 = C_2 +$ {package observer; object SubjectImpl singleton ... { contents = list_test_nil; }}

Fig. 6. Example trace of `observer`

Theorem 1 (Full Abstraction). *For all well-typed components C_1 and C_2, we have*

$$\mathsf{Traces}(C_1) = \mathsf{Traces}(C_2) \quad \textit{if and only if} \quad C_1 \simeq C_2$$

The proof of this theorem is non-trivial and unfortunately too long to present in full here. It breaks in to two parts, soundness and completeness, which together are sufficient to show full abstraction. For the remainder of this section we will outline the proof technique.

6.1 Soundness of Trace Inclusion for Testing

We show the soundness of our model with respect to testing equivalence, that is,

$$\mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2) \quad \text{implies} \quad C_1 \lesssim C_2.$$

To demonstrate this we observe that the traces are defined in such a way as to guarantee that every (internal) interaction between C_i and a test component can be decomposed in to complementary traces (traces which are identical except for the reversal of the ! and ? annotations) and conversely that pairs of complementary traces can also be (re)composed to obtain an internal reduction. This is a not a straightforward property to express as the result of an active thread in C_1 and C is typically an intertwining of code from each component. To describe the possible states of the system we overload the notation $\between$ to define how to *merge* complementary states of the labelled transition system.

Define $\bar{t}_1 \to \bar{u}_1$ and $\bar{t}_2 \to \bar{u}_2$ to be *mergeable* for t as:

- ε and ε are mergeable for Object.
- If $\bar{t}_2 \to \bar{u}_2$ and $\bar{t}_1 \to \bar{u}_1$ are mergeable for u
  then $\bar{t}_1 \to \bar{u}_1$ and $t \to u, \bar{t}_2 \to \bar{u}_2$ are mergeable for t.

Define $(C_1 \vdash E : t \rhd \bar{\mathcal{E}}_1 : \bar{t}_1 \to \bar{u}_1)$ and $(C_2 \vdash \mathsf{block} \rhd \bar{\mathcal{E}}_2 : \bar{t}_2 \to \bar{u}_2)$ are *mergeable* when

- C_1 and C_2 are linkable and $\bar{t}_1 \to \bar{u}_1$ and $\bar{t}_2 \to \bar{u}_2$ are mergeable for t

We define the partial merge $\bar{\mathcal{E}}_1 \between \bar{\mathcal{E}}_2$ of context stacks as

$$\varepsilon \between \varepsilon = \cdot$$
$$\bar{\mathcal{E}}_1 \between (\mathcal{E}_2, \bar{\mathcal{E}}_2) = (\bar{\mathcal{E}}_2 \between \bar{\mathcal{E}}_1)[\mathcal{E}_2]$$

When Σ_1 and Σ_2 are mergeable we define $\Sigma_1 \between \Sigma_2$ as the state given by:

$$(C_1 \vdash E_1 : t_1 \rhd \bar{\mathcal{E}}_1 : \bar{t}_1 \to \bar{u}_1) \between (C_2 \vdash \mathsf{block} \rhd \bar{\mathcal{E}}_2 : \bar{t}_2 \to \bar{u}_2) = (C_1 \between C_2 \vdash (\bar{\mathcal{E}}_1 \between \bar{\mathcal{E}}_2)[E_1])$$

Proposition 4. *If* $\vdash \Sigma_1 : \mathsf{state}$ *and* $\vdash \Sigma_2 : \mathsf{state}$ *are mergeable and* $\Sigma_1 \between \Sigma_2 = (C \vdash E)$, *then* $\vdash C : \mathsf{component}$ *and* $C \vdash E : \mathsf{Object\ in} * .$

We write $\bar{a}^{-1}$ for the trace $\bar{a}$ with the input and output annotations reversed. We also define the *external ordering*, $C \sqsubseteq^{ext} C'$ as the preorder on components generated by $C \sqsubseteq^{ext} C + \mathsf{object}\ t\ v;$. Note that whenever $C \sqsubseteq^{ext} C'$ then C' differs only in that it may contain more external interface types for object definitions.

Proposition 5 (Trace Composition/Decomposition). *If* Σ_1 *and* Σ_2 *are mergeable such that* $\Sigma_1 \between \Sigma_2 = (C \vdash E)$ *then*

1. *If* $\Sigma_1 \xrightarrow{\bar{a}} \Sigma_1'$ *and* $\Sigma_2 \xrightarrow{\bar{a}^{-1}} \Sigma_2'$ *then* $(C \vdash E) \to^* (C' \vdash E')$ *where either*
   - Σ_1' *and* Σ_2' *are mergeable such that* $\Sigma_1' \between \Sigma_2' = (C'' \vdash E')$ *with* $C' \sqsubseteq^{ext} C''$, *or*
   - Σ_2' *and* Σ_1' *are mergeable such that* $\Sigma_2' \between \Sigma_1' = (C'' \vdash E')$ *with* $C' \sqsubseteq^{ext} C''$.
2. *If* $(C \vdash E) \to^* (C' \vdash E')$ *then there exists* $\bar{a}$ *such that* $\Sigma_1 \xrightarrow{\bar{a}} \Sigma_1'$ *and* $\Sigma_2 \xrightarrow{\bar{a}^{-1}} \Sigma_2'$ *where either*
   - Σ_1' *and* Σ_2' *are mergeable such that* $\Sigma_1' \between \Sigma_2' = (C'' \vdash E')$ *with* $C' \sqsubseteq^{ext} C''$, *or*
   - Σ_2' *and* Σ_1' *are mergeable such that* $\Sigma_2' \between \Sigma_1' = (C'' \vdash E')$ *with* $C' \sqsubseteq^{ext} C''$.

Theorem 2 (Soundness of traces for may testing). *For compatible* C_1 *and* C_2 *which accept* `System`, *if* $\mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2)$ *then* $C_1 \lesssim C_2$.

Proof: (Sketch) Suppose that $\mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2)$ and also suppose that C is a testing component such that $C \between C_1$ prints the message "Hello" during evaluation. We can use Trace Decomposition on the interaction between C and C_1 which caused this string to be produced. This gives a pair of complementary traces. Now, because $\mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2)$ we also know that C_2 must perform the same traces as C_1. Therefore, when we link C with C_2, because these components can respectively perform the given pair of complementary traces, these may be re-composed using Trace Composition to give an internal evaluation of $C \between C_2$ which will also print the message "Hello". This can be done for any testing component and any message. So this serves to demonstrate that $C_1 \lesssim C_2$. □

6.2 Completeness

The converse property, completeness,

$$C_1 \lesssim C_2 \quad \text{if and only if} \quad \mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2)$$

also relies on the Trace Composition/Decomposition property (Proposition 5). In addition to this though we also need to show a definability result which states that for every (odd length) trace from a well-typed component we can find a component and expression which will exhibit this trace, and only this trace (up to renaming of fresh names).

Proposition 6 (Definability). *If we have* $\vdash C : \mathsf{component}$ *and (for* $\bar{a}$ *of odd length)* $(C \vdash \mathsf{block} \rhd \varepsilon : \varepsilon) \stackrel{\bar{a}}{\Longrightarrow} \cdots$ *then we can find* C'*,* E *such that* $\vdash C' : \mathsf{component}$*,* C *and* C' *are linkable,* $C' \vdash E : \mathsf{Object}\ \mathsf{in} * $ *and* $(C' \vdash E : \mathsf{Object} \rhd \varepsilon : \varepsilon) \stackrel{\bar{b}}{\Longrightarrow} \cdots$ *(for* $\bar{b}$ *of odd length) if and only if* $\bar{b} \equiv \leq \bar{a}^{-1}$.

Theorem 3 (Completeness of traces for may testing). *For compatible* C_1 *and* C_2*, if* $C_1 \lesssim C_2$ *then* $\mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2)$.

Proof: (Sketch) Suppose that $C_1 \lesssim C_2$ and also suppose that C_1 has a trace $\bar{a}$. We can suppose (wlog) that $\bar{a}$ is even length. We must show that C_2 also has this trace. We do this by first applying our definability result to the complement of $\bar{a}$ extended with a visible action of an outgoing call to `System.print` with message, "Hello", say. This yields a component `C_def`. Given this, we use Trace Composition with $\bar{a}$ and its complement by linking `C_def` and C_1 to yield an internal evaluation which will ultimately print "Hello". Because, $C_1 \lesssim C_2$ and because `C_def` acts as a test, we know that `C_def` $\between C_2$ must also evaluate and eventually print the message "Hello". We use Trace Decomposition to split this internal evaluation into separate traces to see that C_2 must perform *some* trace complementary to that of `C_def`. But, given that `C_def` performs the unique (up to $\equiv$) trace $\bar{a}^{-1}$, we must have that C_2 has the trace $\bar{a}$ also. □

7 Conclusions and Further Work

We have described a novel core Java-like language, Java Jr., which allows package and class-based definitions of object systems. The language is specifically designed to be semantically clean by using the packaging system to enforce all cross-package interaction to be limited to method invocation and return. We provided Java Jr. with a static type system, whose types act as interfaces to components for building large systems and we presented a simple notion of linking for plugging well-typed components together. This notion of linking was also used to give an extensional definition of a component's suitability for substitution with other components from a structural, or static point of view. We proceeded to develop an extensional definition of a component's suitability for substitution from a behavioural, or dynamic point of view, drawing on a body of work in the literature on process testing [15]. Importantly, our trace semantics for Java Jr. components turn out to capture the notion of behavioural testing precisely.

This Full Abstraction property is a major result of this work and is the first such result for a class-based object language with packages and subtyping.

Although the core language only supports a limited number of Java features, it already has enough power to encode some of the more sophisticated control-flow operations and, to some extent, is robust enough to allow the addition of extra features without disrupting the higher-level semantics, as long as these new features do not create new cross-package interaction. For instance, it is straightforward, but slightly cumbersome to include primitive types and constants in Java Jr. These would not seriously affect the validity of the full abstraction theorem. We also anticipate that Java exceptions could readily be included within our framework, although this would require exception interfaces in addition to Java's exception classes, if we want Java Jr. exceptions to be thrown and caught across package boundaries.

Unfortunately, some of Java's features have significant impact on our model: in particular, explicit downwards typecasts and concurrency. Downwards casts affect our trace semantics in a fairly significant way. At present we maintain strict public interfaces to classes and only release objects at given interfaces. This type security is maintained in Java Jr. as there is no possibility for code to use a received object at any lower type. This is reflected in the trace semantics by recording lists of interface types at which component and environment object names have been leaked. Allowing downwards casts would enable code in a given package to receive objects declared in a different package and discover their private types. This breaks the *programming to an interface* discipline of Java Jr.

Similarly, the trace model is built around the notion of a single thread of control. The straightforward alternation of control between component and environment in the trace semantics is a direct consequence of this. Fortunately, introducing named threads and providing an interleaved trace model is achievable. Earlier work of ours [17] provides such a model for a small concurrent object-based calculus. The extra complications of classes and subtyping present are unlikely to affect the integration of the concurrent thread model within Java Jr.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. S. Abramsky and L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105:159–267, 1993.
3. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer, 1999.
4. I. Attali, D. Caromel, and M. Russo. A formal executable semantics for Java. In *Proc. Formal Underpinnings of Java*, 1998.
5. A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *J. Functional Programming*, 2005. To appear.
6. G.M. Bierman, M.J. Parkinson, and A.M. Pitts. An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2003.
7. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.
8. L. Cardelli and A. Gordon. Mobile ambients. In *Proc. Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

9. S. Drossopoulou and S. Eisenbach. Towards an operational semantics and proof of type soundness for Java. In *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 41–82. Springer-Verlag, 1999.
10. M. P. Fiore, E. Moggi, and D. Sangiorgi. A fully-abstract model for the pi-calculus. In *Proc. IEEE Logic in Computer Science*, page 43. IEEE Computer Society, 1996.
11. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. ACM Principles of Programming Languages*, pages 171–183, 1998.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
13. M. J. C. Gordon. *Experimental Programming Reports*. PhD thesis, School of AI, University of Edinburgh, 1973.
14. J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley, 2000.
15. M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
16. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
17. A.S.A Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proc. IEEE Logic in Computer Science*, pages 101–112. IEEE Computer Society Press, 2002.
18. P.J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Communications of the ACM*, 8(2):89–101, 1965.
19. J. McCarthey. Towards a mathematical science of computation. In C.M. Popplewell, editor, *Information Processing 1962*, pages 21–28, 1963.
20. J. McCarthy. A formal description of a subset of algol. In *Proc. IFIP WG Formal Language Description Languages for Computer Programming*, pages 1–12. North Holland, 1966.
21. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
22. R. Milner. *Communication and mobile systems: the π-calculus*. Cambridge University Press, 1999.
23. J. H. Morris. *Lambda-calculus models of programming languages*. PhD thesis, MIT, 1968.
24. T. Nipkow. Jinja: Towards a comprehensive formal semantics for a Java-like language. In *Proc. Marktobderdorf Summer School*. IOS Press, 2003. To appear.
25. G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
26. G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
27. M. Tofte R. Milner and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
28. J. Reppy. *Higher-Order Concurrency*. PhD thesis, Cornell University, 1992. Technical Report TR 92-1285.
29. D. Sangiorgi and D. Walker. *The pi-calculus: A Theory of mobile processes*. Cambridge University Press, 2001.
30. T. B. Steel. A formalization of semantics for programming language description. In *Proc. IFIP WG Formal Language Description Languages for Computer Programming*, pages 25–36, 1969.
31. G.J. Sussman and Jr G.L. Steele. Scheme:an interpreter for extended lambda-calculus. Technical Report Memo 349, MIT AI Lab, 1975.
32. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Author Index